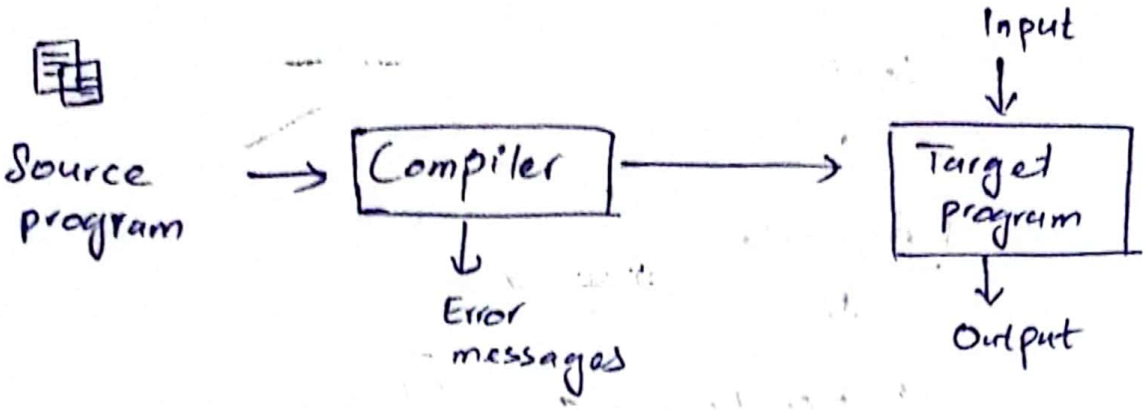
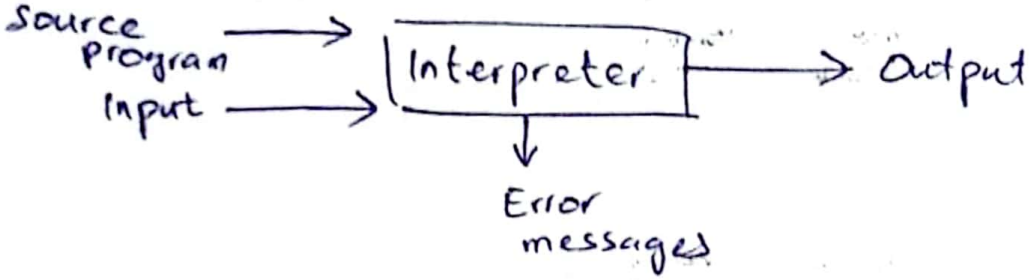
CSC441 - Compiler Design

Prepared by: Mr. Aoun_Haider
fa21-bse-133@cuilahore.edu.pk

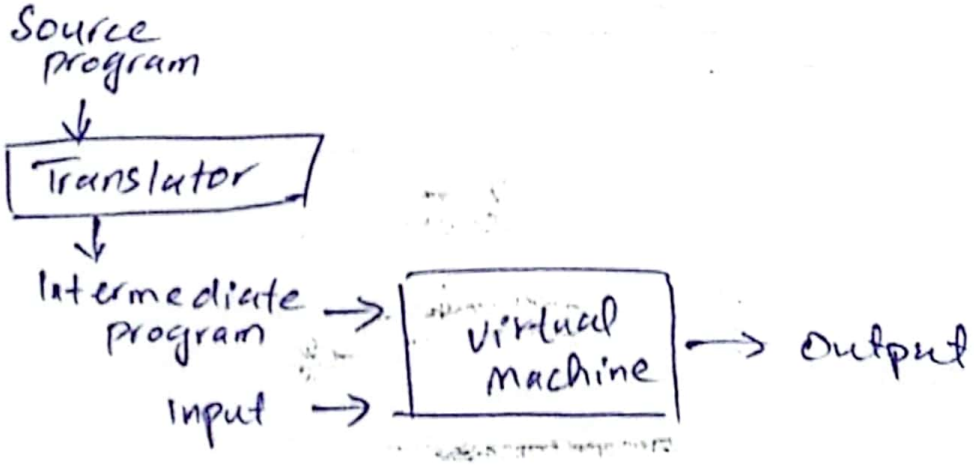
(1) Compiler:- Translation of source program to target program.



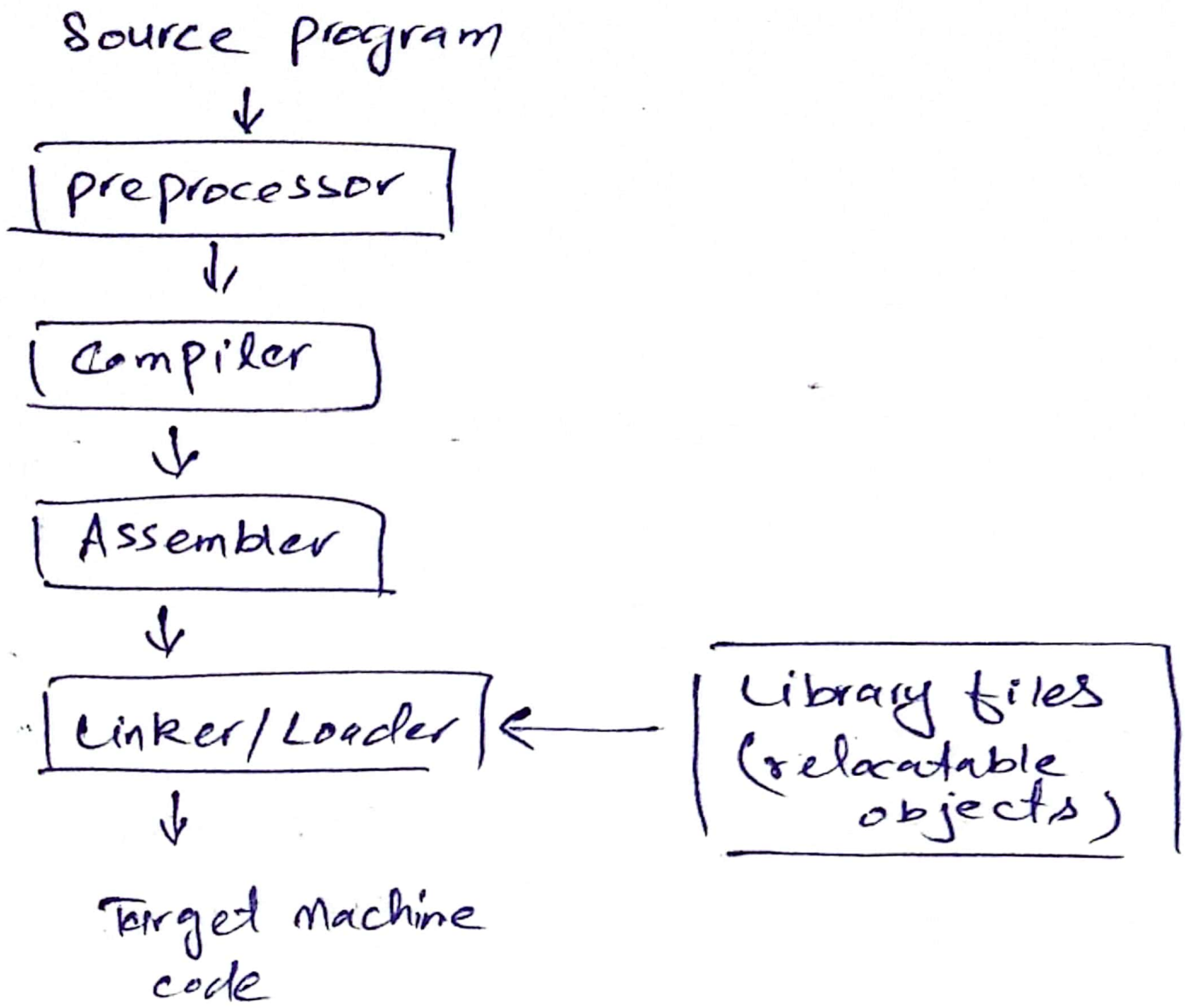
(2) Interpreters:
- line by line compilation
- performing the operation implied by source program



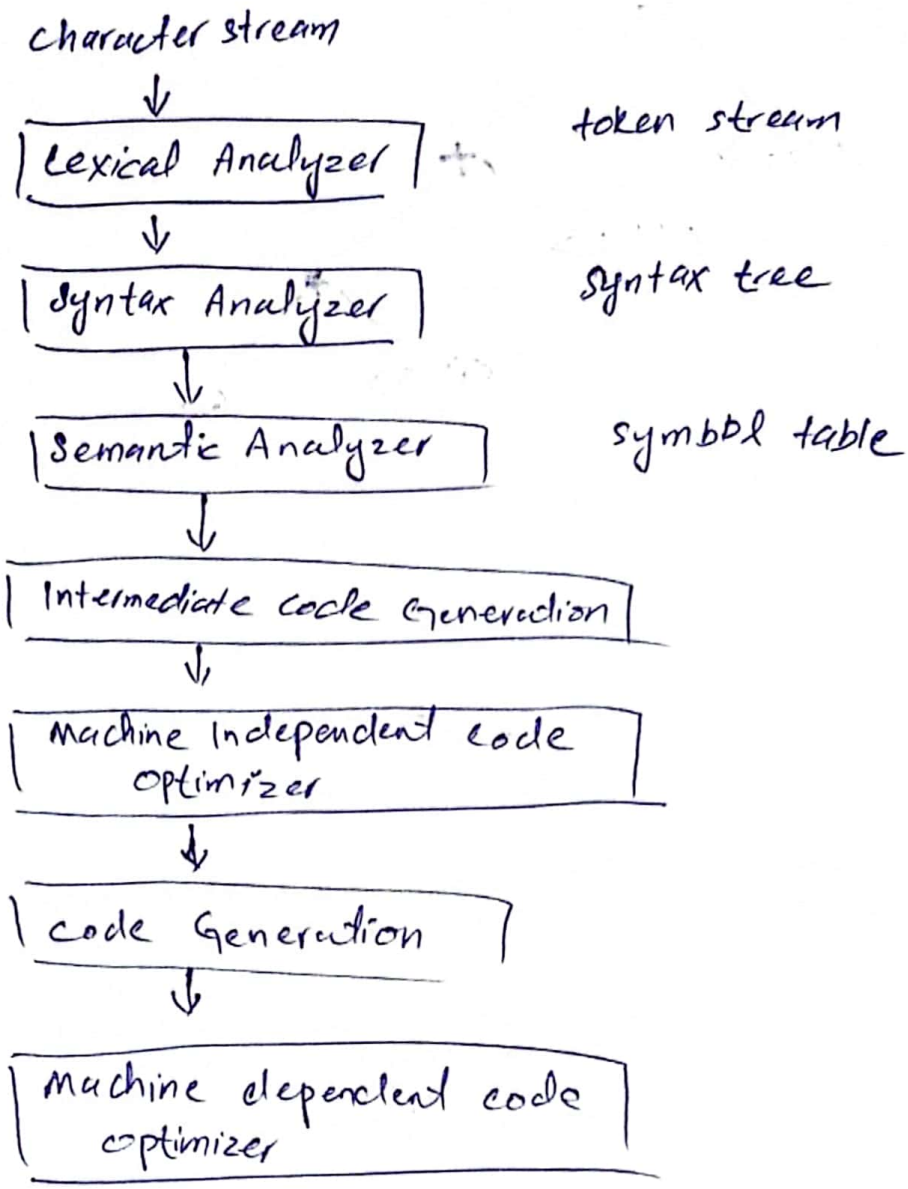
(3) Hybrid:



Language processing system:



phases of Compiler:



Example:

position = initial + rate x 60 ;

① Lexical Analyzer
- Divide each section into tokens and assign 'id' of symbol table

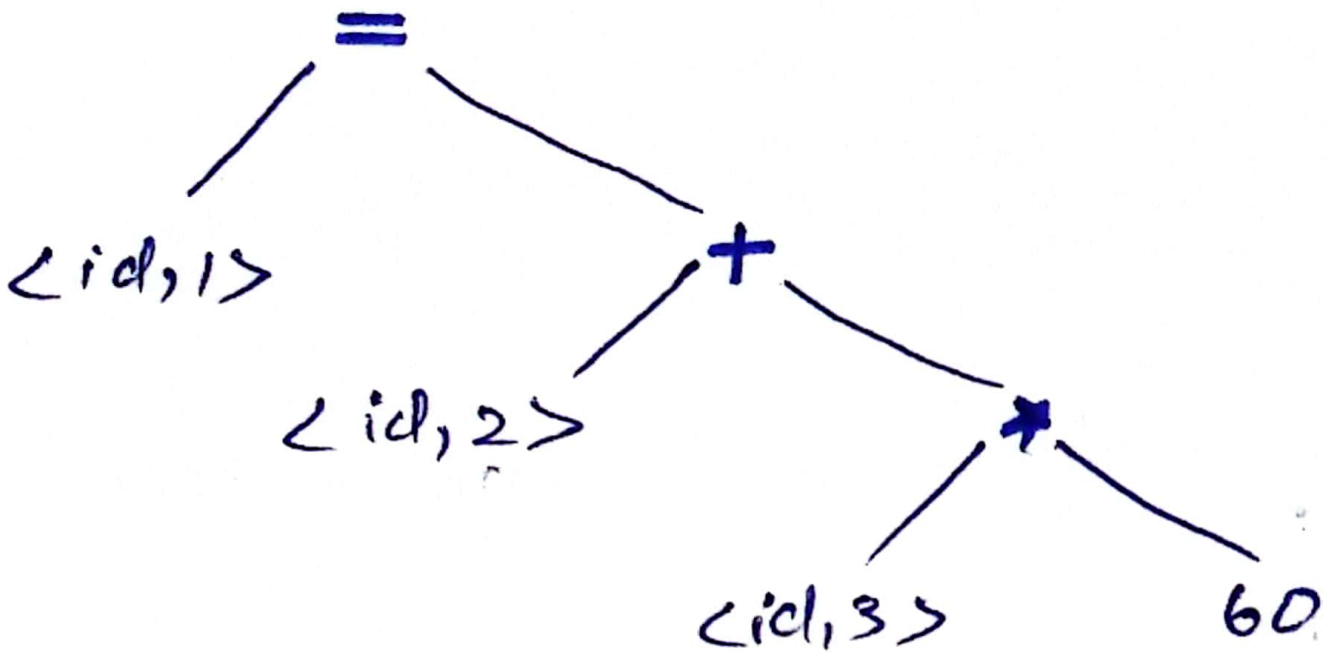
position = <id, 1>
initial = <id, 2>
rate = <id, 3>

<id, 1> <=> <id, 2> <+> <id, 3>
<*> <60>

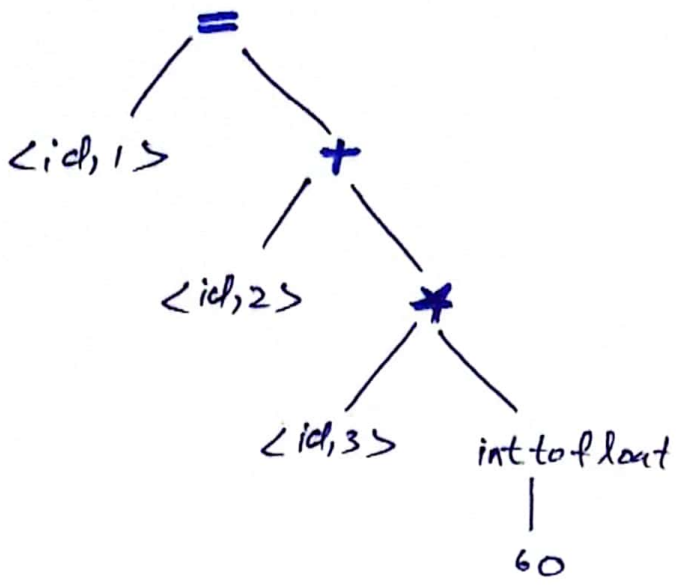
Symbol Table

1	position	...
2	initial	...
3	rate	...

Syntax Analyzer:



3) Semantic Analyzer:



Note: In this case, semantic analyzer has performed type coercion (implicit type casting) due to the type of position 'float' because int and float are different types and type coercion is required to perform operation.

4) Intermediate Code Generation:

```
t1 = intoffloat(60)
t2 = id3 x t1
t3 = id2 * t2
id1 = t3
```

5) Machine independent code optimizer:

```
t1 = id3 x 60.0
id1 = id2 + t1
```

5) Code Generation:

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Types of compiler:

Single pass	Multipass
1) Compiler visits the source code not multiple times	compiler visits the source code multiple times & observe a specific portion each turn
2) Require less memory	More memory requirement
3) Fast	Slower
4) makes less optimal code	makes more optimal code
5) Detects error in linear fashion and require correction of single error every time	Detect errors back to back

Contradiction:

- Compiler is multipass & interpter is single pass (not always true)
- interpreter is used for dynamically typed languages like python while ~~compiler~~ compiler is used for dynamically typed languages.

Statically

Grouping

1. Front End → analysis (machine independent)
2. Back End → synthesis (Machine dependent)

- lexical,
- Syntax,
- Semantic,
- Intermediat code

- Code Generation
- optimization (Register allocation, Instruction scheduling)

① Lexical Analysis: (Scanner, Lexer, tokenizer)

- identifier: defined by programmer i.e variable name
- keyword: language reserved words ('if', 'else', 'for')
- number/constant/literal: 0, 1, 2, ... 9
- operator: Separator or punctuators {, (,), }

> Analysis task:

① Tokenization

② Error messages → Exceeding strings (long variable names)
→ illegal character (unending multiline comment)

③ Eliminate comments & whitespaces, tabs, \n, \t

NFA or DFA are used in this phase

Example:

① `printf("i=%d, &i=%x", i, &i);`

1 2 3

10 tokens

② `main() {` /* Cui lahore */ → comments are treated as blank characters

`a = b+++---++++==;`

`printf("Aoun: %d", a);`

`}`

"Look ahead problem" i=if || = or ==

- Ambiguity where a token ends and next token starts

Steps:

→ Scan the character stream left to right

→ Identify each token

→ Identify each tokens' category

Regular Expression

Σ = set of character + = or

* = zero or more occurance

+ = one or more occurance

$A^+ = AA^*$ set of all alphabets

(1) string of letter or digits starting with letter

digit = '0' + '1' + '2' + ... + '9'

letter = 'a' + 'b' + 'c' + ... + 'z'

= letter (letter + digit)*

(2) phone number: (650)-723-3232

x = digit^3

y = digit^4

'(' x ')' - ' x '-' y

(3) Unsigned digit -2.3e-3

digit = '0' + '1' + ... + '9'

digits = digit+

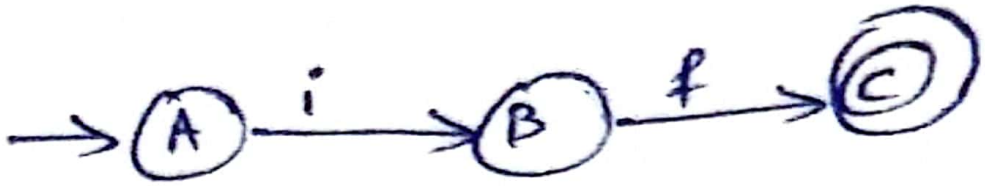
fraction = ('.' digits) + ϵ

exp = ('e' ('+' + '-' + ϵ) digits) + ϵ

number = digits fraction exp

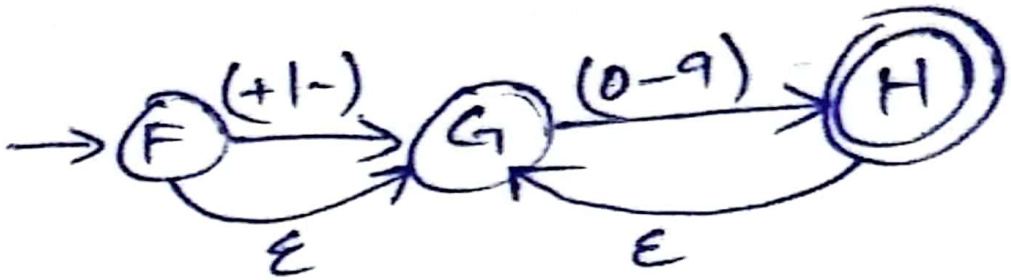
Tokens' DFA

- if :



- identifier:

- Integer:



ε = space, no input

Remaining after 1 page

Symbol Table

- produced by analysis phase (front end) and consumed by ~~analysis~~ synthesis phase (back end)
- Data structure maintained by compiler to store instance of variable, objects, classes, interface etc.
- Tasks performed by each phase regarding symbol table

(1) Lexical : create entries for identifier

(2) Syntax: adds information regarding attributes

(3) Semantic: Using available information, updates the symbol table.

(4) Intermediate code : Available info helps in adding temporary variables information

(5) Optimization : Available info is used in machine dependent optimization

(6) Target code: Generate target code using address info of identifiers

- Sample symbol table: int rate = 133;

Name	Type	Size	Dimension	Line of declaration	Line of usage	Address
rate	int	4	0	5	7→9→10→NULL	0X3B076F

- Size of symbol table must be considered

too small : no space to store more entries dynamically

too large : wastage of space

So, symbol table must claim dynamic memory

Non-Block structured language: Fortran

Each variable must be declared once in entire program

Operations: insert(), lookup()

Block structured:

- Variables can be declared in multiple blocks / scope / locality

- Operations:

  Insert(), Lookup(), set(), Reset()

All latest languages are block structured

Quiz:

① Which one is not intermediate representation of source program?

a) 3-address code b) Abstract syntax tree

c) Symbol table d) Control Flow Graph (CFG)

② Access time of symbol table will be O(logn) if implemented via

a) Linear List

b) B. Search Tree

c) Hash table

d) None

Explanation:

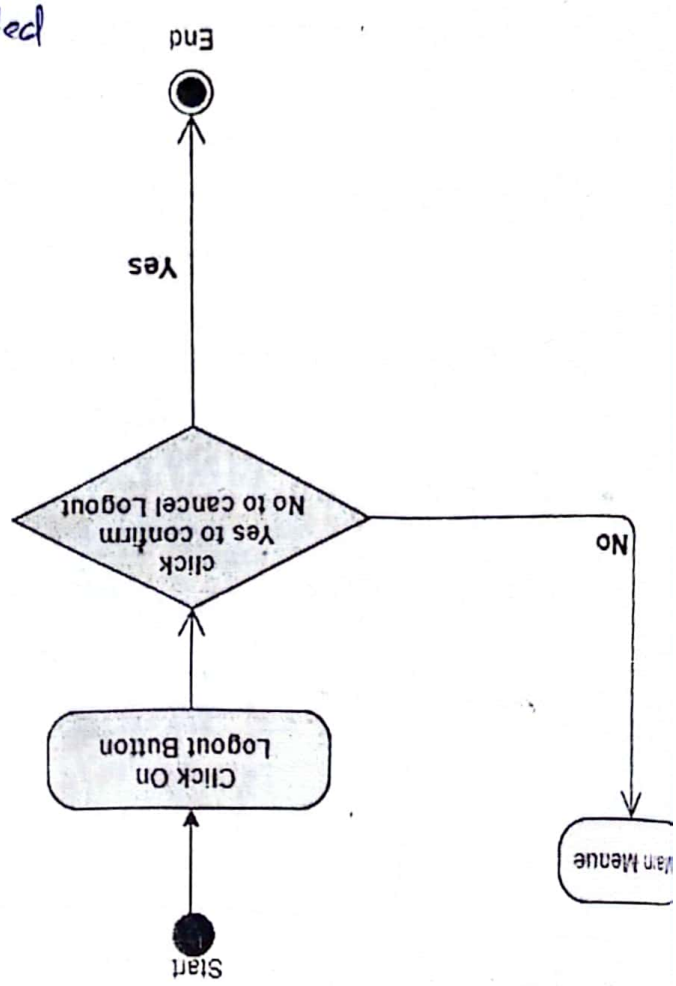
→ Linear list
  ordered O(n)
  → linked
  unordered

→ Tree

$O(\log_m n)$

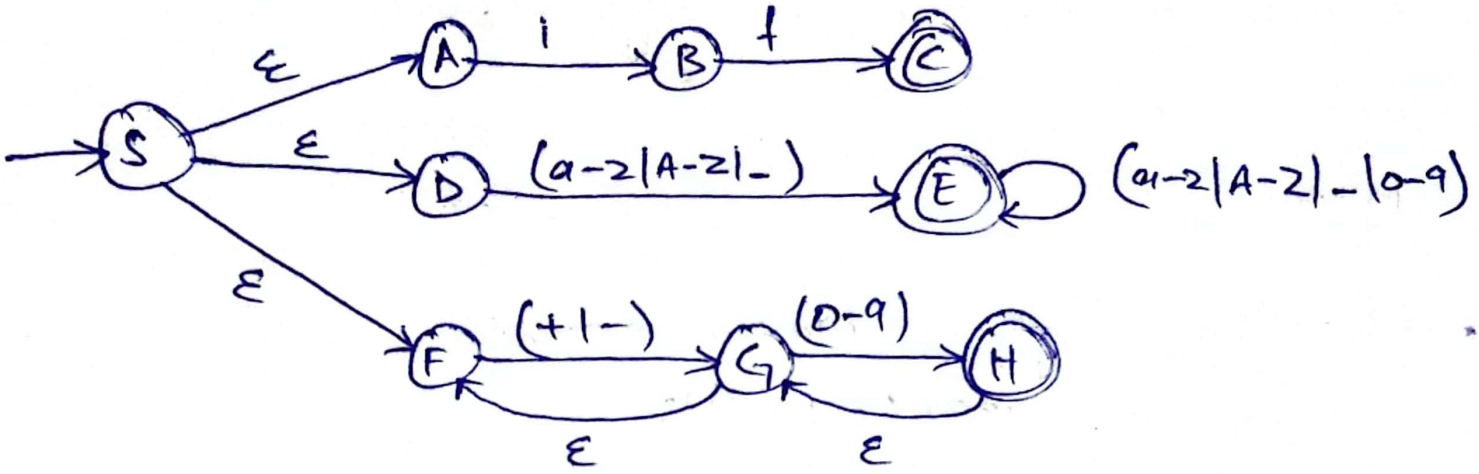
→ Hash table

- if no collision O(1)
- otherwise ≈ O(n)

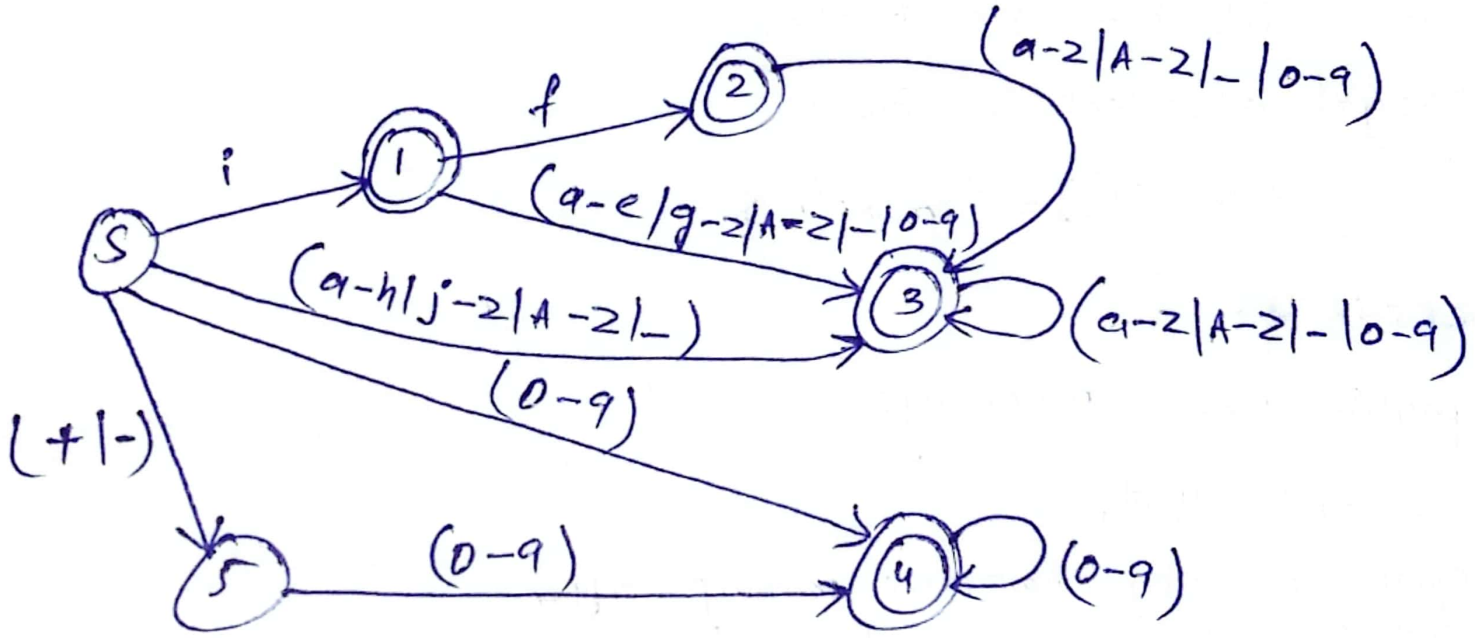


NFA:

- Let say we want to recognize if, identifier & digits we need to combined all the DFAs and introduced a new state 'S'



DFA:



Errors & Error Recovery in Lexical Analysis:

- **Lexical Errors:**
  - Identifiers are too long
    C allows variable name of length = 247
    C++ " = 2048
    python " = 79
  - Exceeding length of numeric constants
    int id = 13345678911;
    if 2 bytes allowed, error!
  - Numeric constants are ill-formed
    int i = 4567$91;
  - Illegal characters addition;
    char x[] = "ABCN"; $

Recovery:

- panic mode recovery:
  int 4rd;
  ignore remaining characters scanning after detecting invalid character
- Transpose of 2 adjacent characters

```
unoin test {          →    Union test {
    int x;                     int x;
    float y;                   float y;
} t1;                      } t1;
```

- Insert missing character

```
it id;   →   int id;
```

- Delete unknown extra character

```
intt id;   →   int id;
```

- Replace a character with another

itt id; → int id;

Example:

0 height = (width + 56) * factor(foo);

- Generate symbol table, count # of tokens, create CFG and syntax tree

① Symbol table:

	Name	type	size	Dimension	Line of declaration	Line of usage	Address
0	height	int	4	0	-	0	-
1	width	int	4	0	-	0	-
2	factor	~~void~~ int	20	0	-	0	-
3	foo	int	4	0	-	0	-

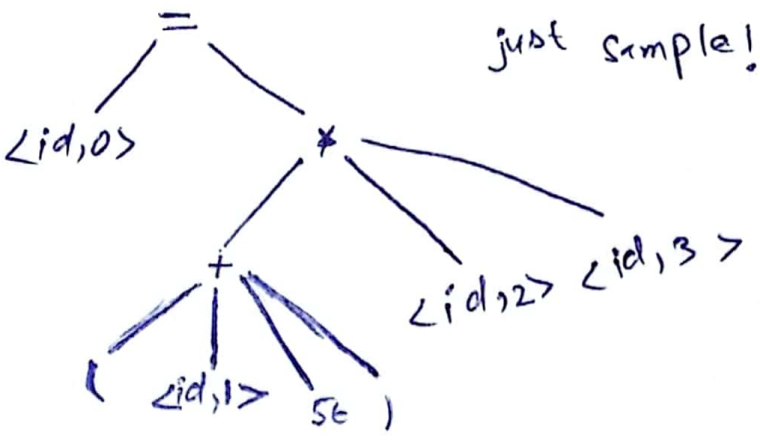
② # of tokens = 13

③ CFG

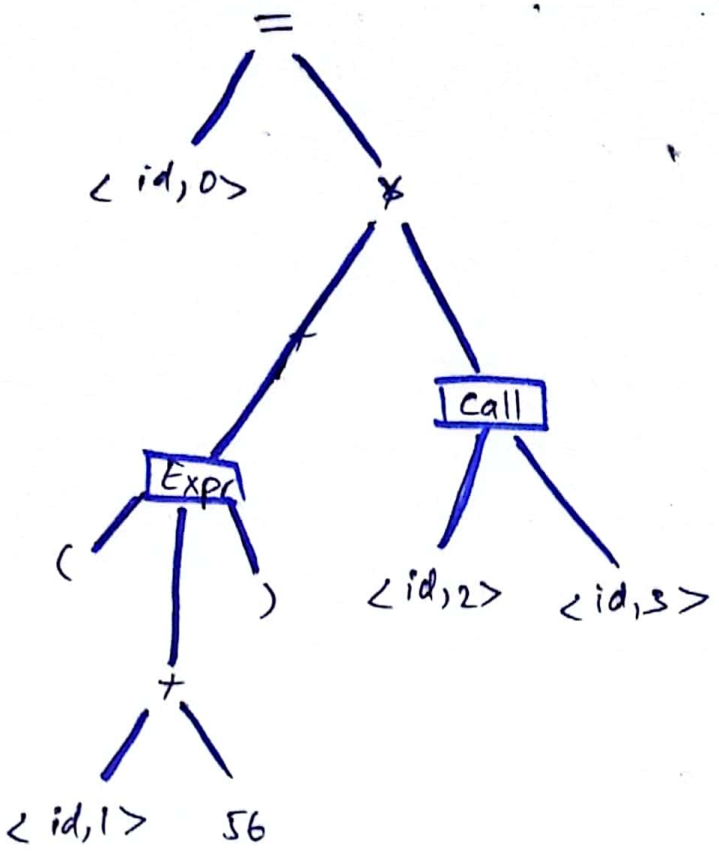
expr → (expr) | expr * expr | int | id(expr) | id

expr = expr

④ Syntax tree



just sample!

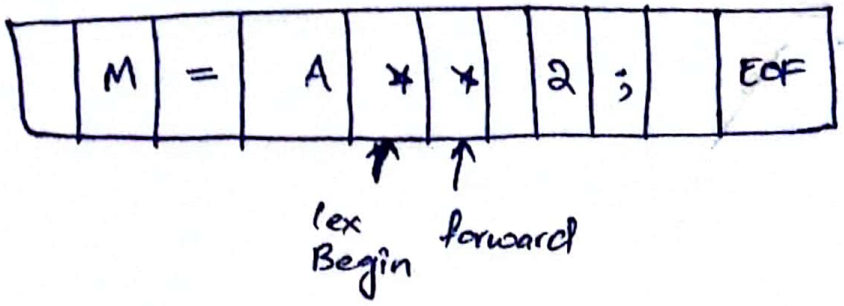


<id,0> <=> <(> <id,1> <+> <56> <)> <*>
<id,2> <(> <id,3> <)> <;>

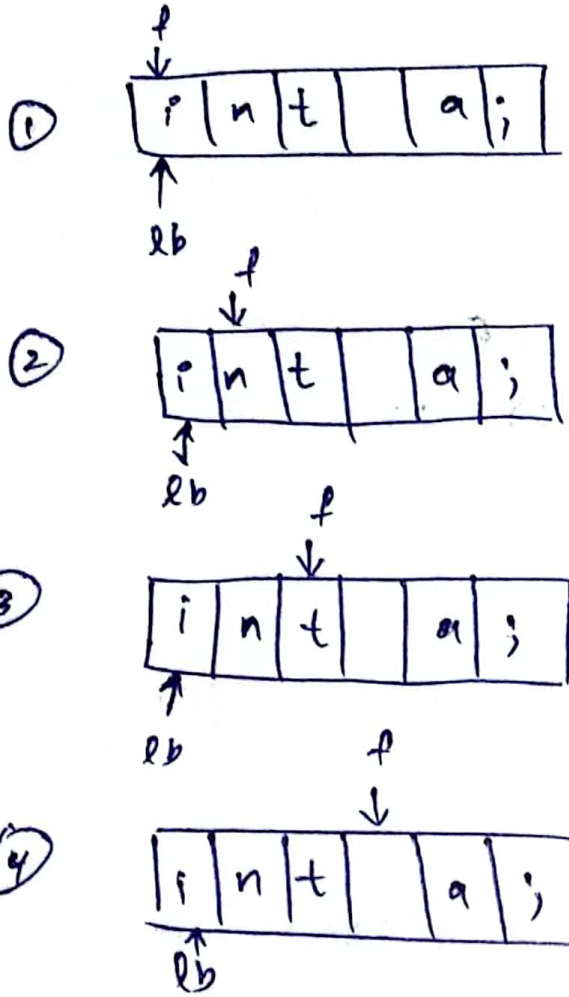
Input buffering:

- Store the source program code in a temperary storage area called buffer for processing
- Use two pointers **lexemBegin** and **foward** in buffer
- Put a sentinal EOF at the end of buffer to indicate end of file

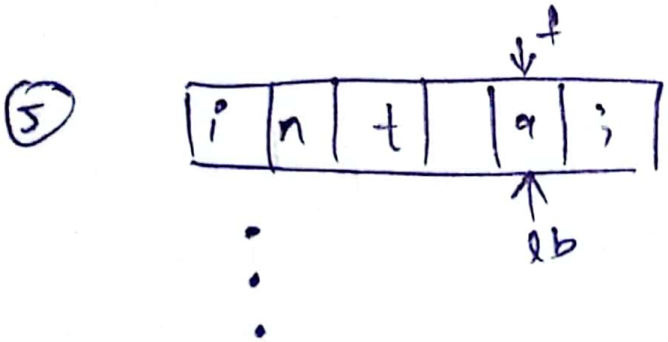
Sometimes, we cannot decide operator by just observing or scanning single character. For example, = vs == or >= or =. gt means we need some mechanism to keep track of next character as well to tick it as legal character (lexeme) That's why two pointers are used.



"lexemeBegin" pointer points at the start of token and forward pointer moves one position right and after detecting a token end, identify it as lexeme or not. The lexBegin will move to the start of next token & so on.



token = "int" => lexeme



reading a single character require read() system call. For reading program of 1000 characters, 1000 system calls are needed.

Buffering helps us to reduce this overhead by reading complete program in a single disk block size memory (4K block size = 4K block memory in RAM) requires only one system call.

Schemes:

① One Buffer:

- Use only one buffer to read input stream
- If input stream size is larger than buffer size, buffer will overflow

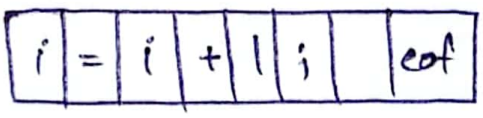
Example: Size of input stream = 500

buffer size = 100

Load 1st 100 characters 0-99, after processing 0-99 characters, override the buffer with 100-299 until complete program is ~~from~~ processed.

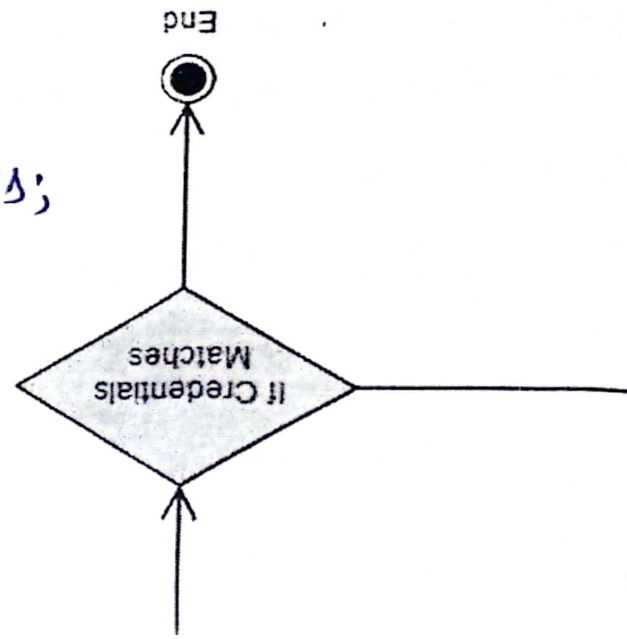
② Two buffer:

- Use 2 buffers to read input stream
- put sentinal character 'eof' at the end of each buffer which is not part of source code.



Double buffering Algorithm

```
switch(*forward++) {
    case eof:
        if (forward is at end of 1st buffer) {
            reload 2nd buffer;
            forward = begining of 2nd buffer;
        }
        else if (forward is at the end of 2nd buffer) {
            reload 1st buffer;
            forward = beginning of 1st buffer;
        }
        else
            terminate lexical analysis;
            break;
    // case of other characters
}
```



String related terms:

① **prefix**: Remove zero or more characters from end of string

banana = ban, banan, ε, banana, ba, bana

② **Suffix**: Remove zero or more characters from start of string

aoun = oun, aoun, un, ε

③ **substring**: By deleting prefix or suffix from string

aoun = ou, ao

④ **subsequence**: Removing zero characters from random position not consecutive

aoun = ou, an, on

Context Free Grammer: CFG

- Way to describe rules of language

Components: CFG = (S,N,T,P)

1. Terminals: Non-replacable
2. Non-terminals: replacable
3. production rule: set of all the rules
4. Starting state 's'

Ambigous grammer: having more than one parse tree

Example:

Consider C code:

```
if (x < 5) {
    b = 2;
}
else {
    b = 1;
}
```

CFG:

stmt → if **expr** then **stmt** |
if **expr** then **stmt** else **stmt** |
ε

expr → term **relop** term | term

term → id | number

digit → [0-9]

digits → digit+

number → digits (. digits)? (E[+|-]? digits)?

letter → [A-Z a-z]

id → letter (letter | letter + '_' | digit)*

if → if

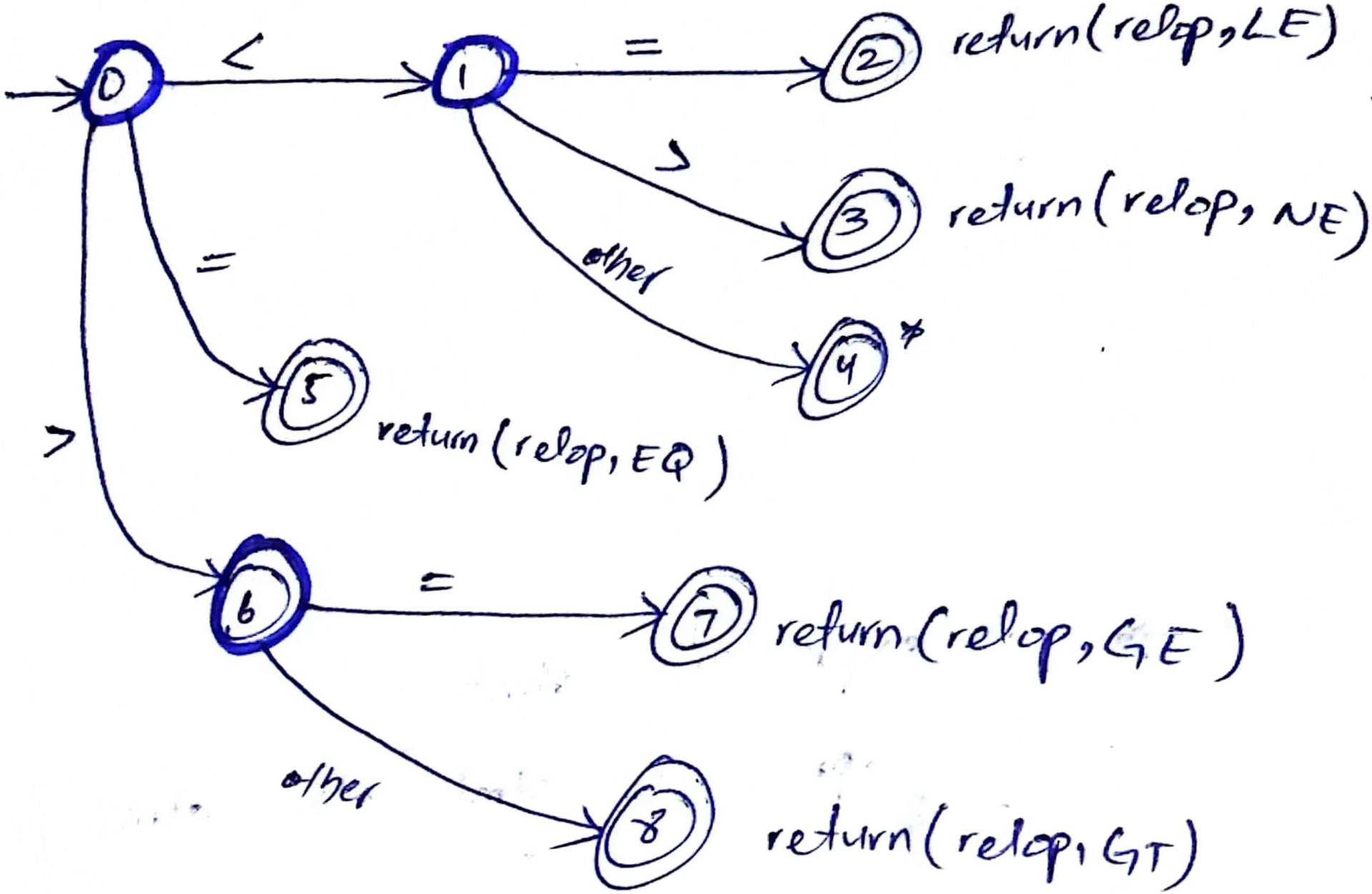
else → else

then → then

relop → < | > | <= | >= | = | <> | !=

stmt = statement
id = identifier
expr = expression
relop = relational operator

Transition Diagram for 'relop'.



How to recognize reserved keywords?

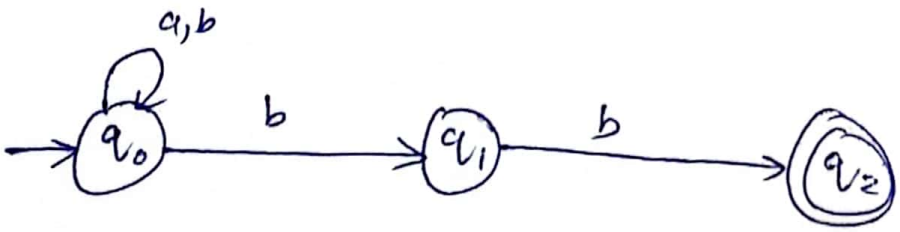
① Store all reserved keywords in symbol table in advance

② Create state transition diagram for each keyword separately

For 'then' keyword in pascal:

Conversion of NFA to DFA:

[Subset Construction Algorithm]



Transition Table:

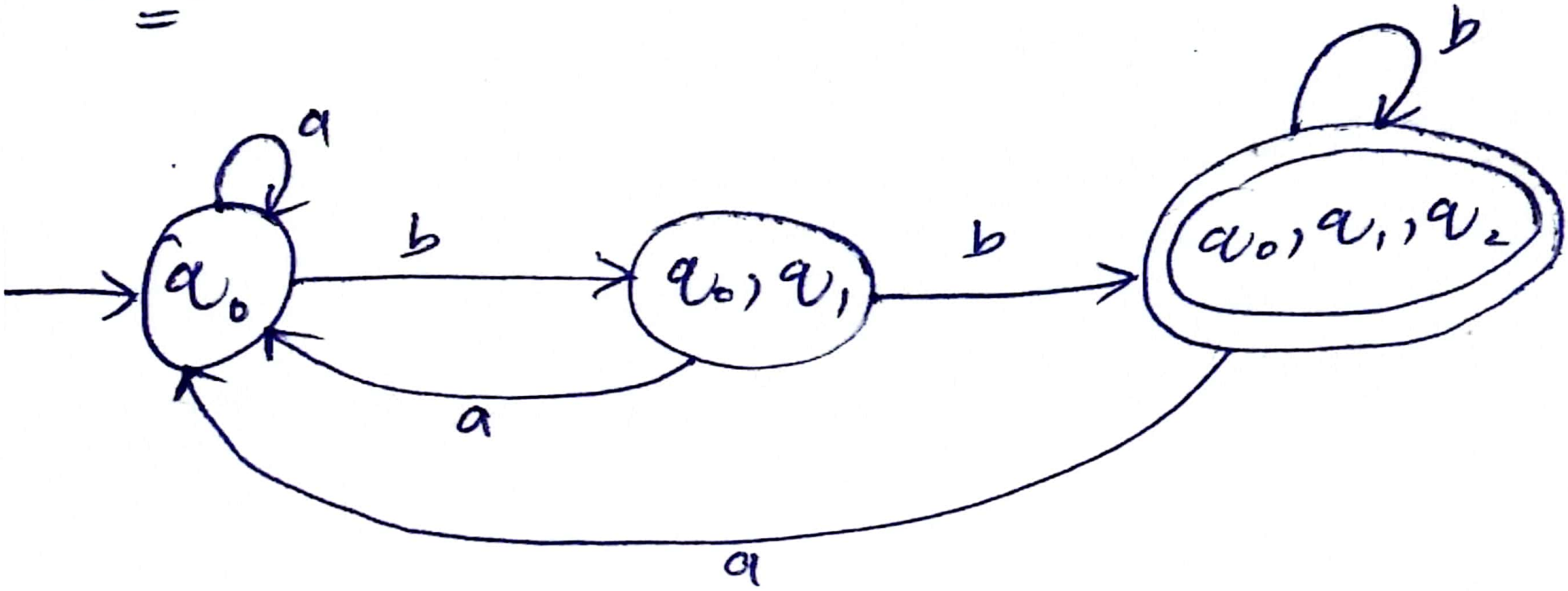
State	a	b
q_0	q_0	q_0, q_1
q_1	∅	q_2
q_2	∅	∅

create q_0, q_1 as new state

state	a	b
q_0	q_0	$\{q_0, q_1\}$
$\{q_0, q_1\}$	q_0	$\{q_0, q_1, q_2\}$
$\{q_0, q_1, q_2\}$	q_0	$\{q_0, q_1, q_2\}$

select final state having 'q_2'

DFA:



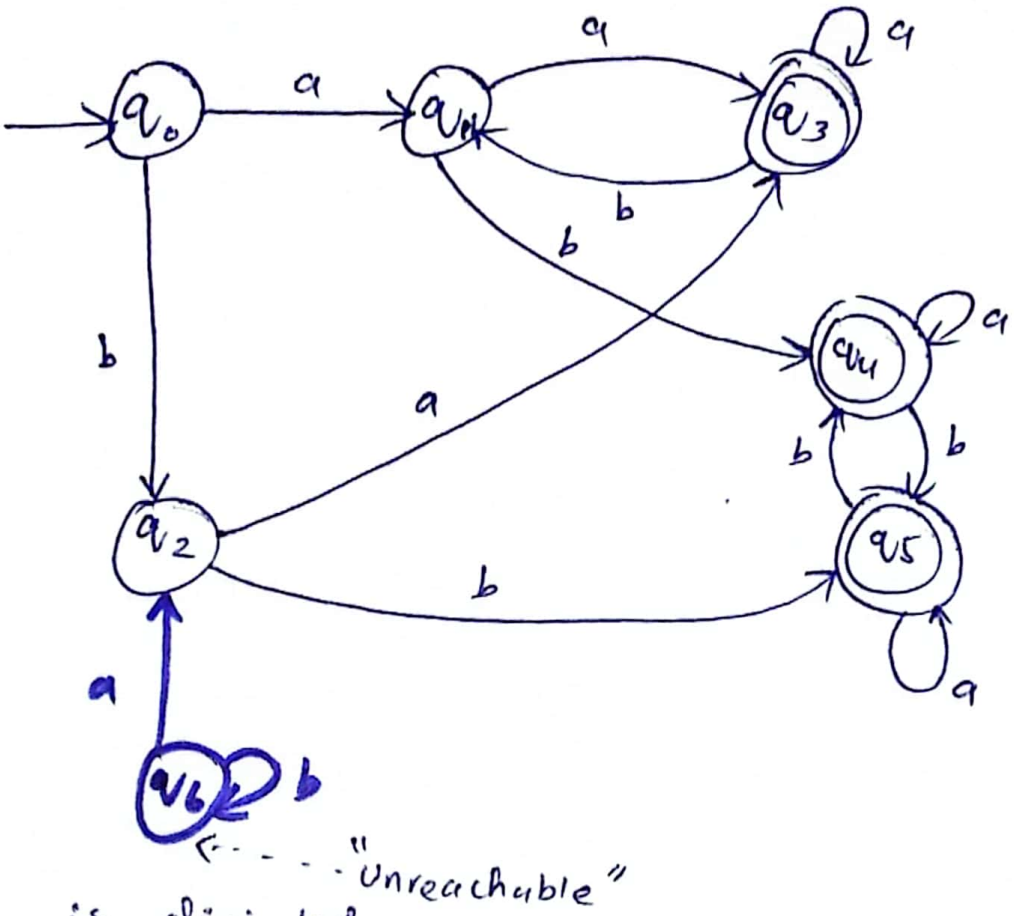
DFA minimization:

① Remove

- Dead states ایسی جہاں سے Termination نہ ہو سکے state
- Unreachable states جہاں نہ جا سکیں

↓

Unable to go from initial state 's' to that specific state



→ q_6 is eliminated

② Create group of terminal and non-terminal states

G_1 G_2

$\{q_0, q_1, q_2\}, \{q_3, q_4, q_5\}$

→ create transition table

→ pick two states from one group & check their destination belong to same group, then pick next two otherwise split/divide into two group

① pick q_0, q_1

S	a	b
q_0	q_1	q_2
q_1	q_3	q_4
q_2	q_3	q_5
→ q_3	q_3	q_1
→ q_4	q_4	q_5
→ q_5	q_5	q_4

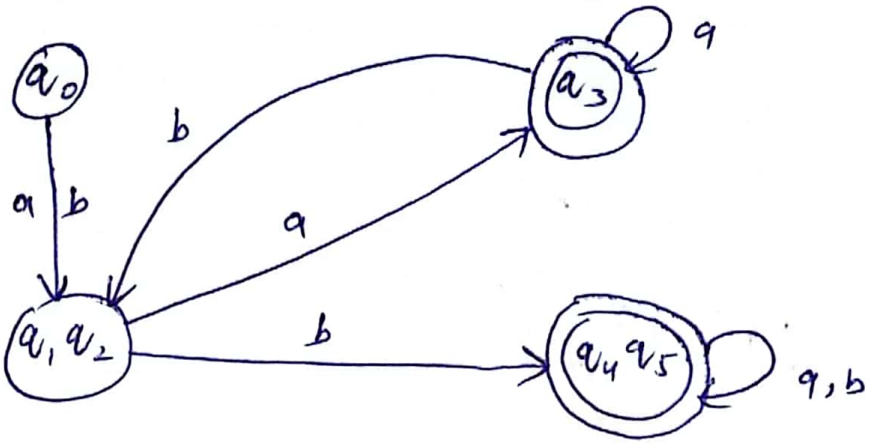
For (q_0, q_1), does these two (q_1, q_3) & (q_2, q_4) belong to same group

non-Terminal

Terminal states

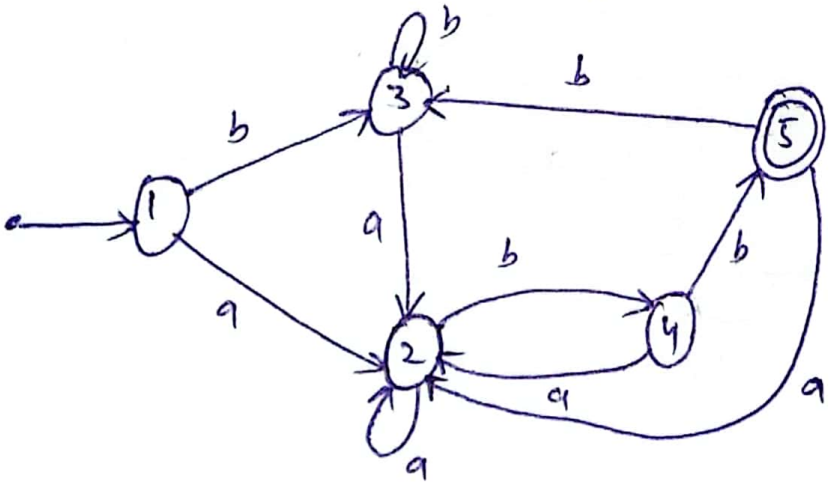
G_1 $[q_0\ q_1\ q_2]$ G_2 $[q_3\ q_4\ q_5]$

G_{11} $[q_0]$ G_{12} $[q_1\ q_2]$ G_{21} $[q_3]$ G_{22} $[q_4\ q_5]$



practise:

Minimize the states of given DFA:



Transition Table

States	a	b
1	2	3
2	2	4
3	2	3
4	2	5
→ 5	2	3

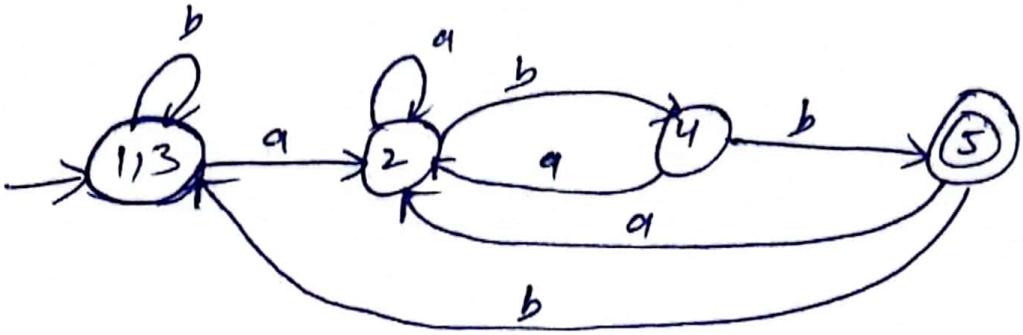
q_1 q_2

[[2 3 4] [5]]

1 4 2

→ (1,2,3) (4) (5)

→ (1,3) (2) (4) (5)



Lex:

- Tool used to create scanner

Syntax:

```
%{
    // declarations &
        preprocessive directives
%}

    // regular expressions

%%

    // methods

%%
int main(){

}
```

- We just need to tell the rules & states, tokens will be generated automatically

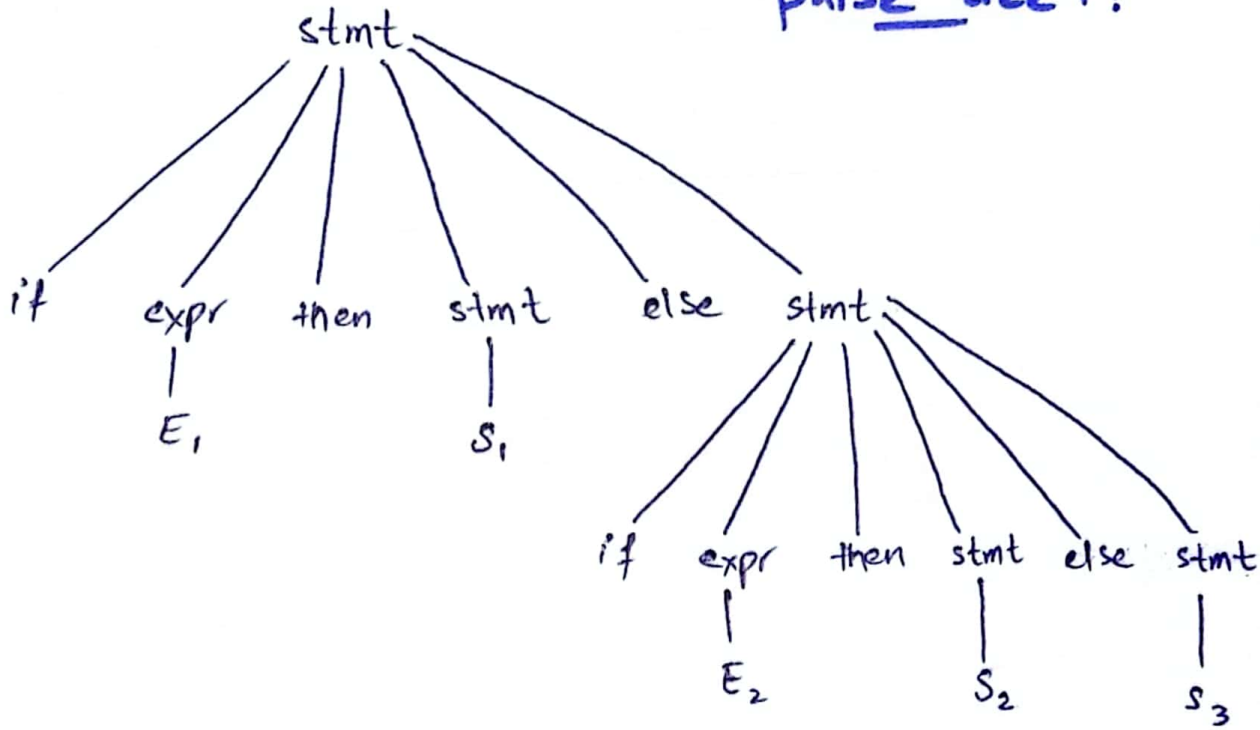
Syntax Analysis (parsing):

- This phase takes stream of tokens from lexical analyzer and generate parse tree via CFG
- The main issue in defining language structure is that CFG must be unambiguous (There must be only one possible parse tree for each input string)

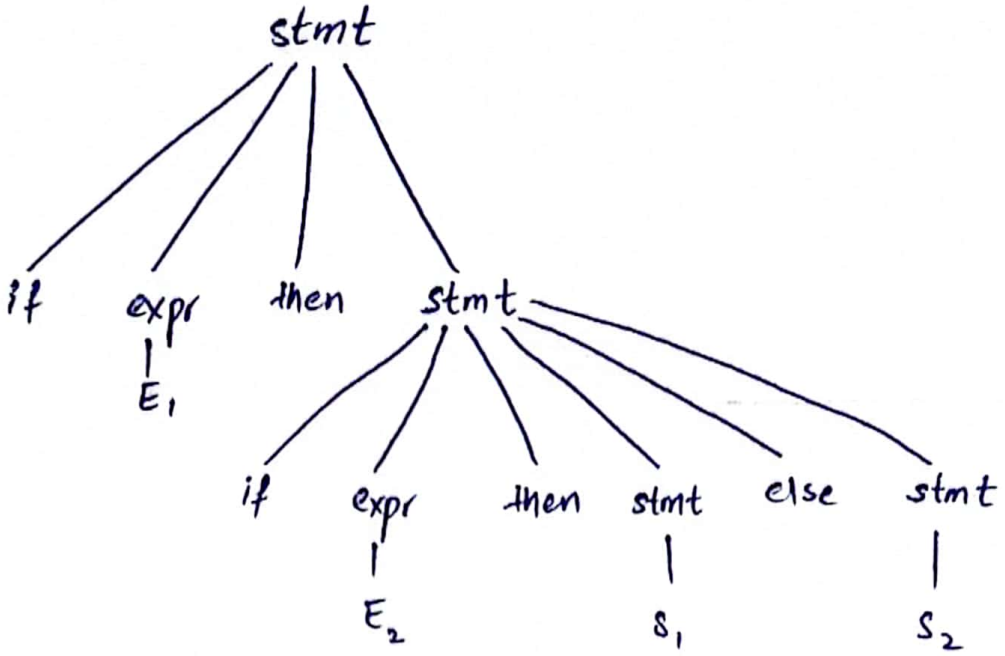
For example:

- stmt → if expr then stmt |
  if expr then stmt else stmt |
  other
- input = if E_1 then S_1 else if E_2 then S_2 else S_3
- possible parse trees:

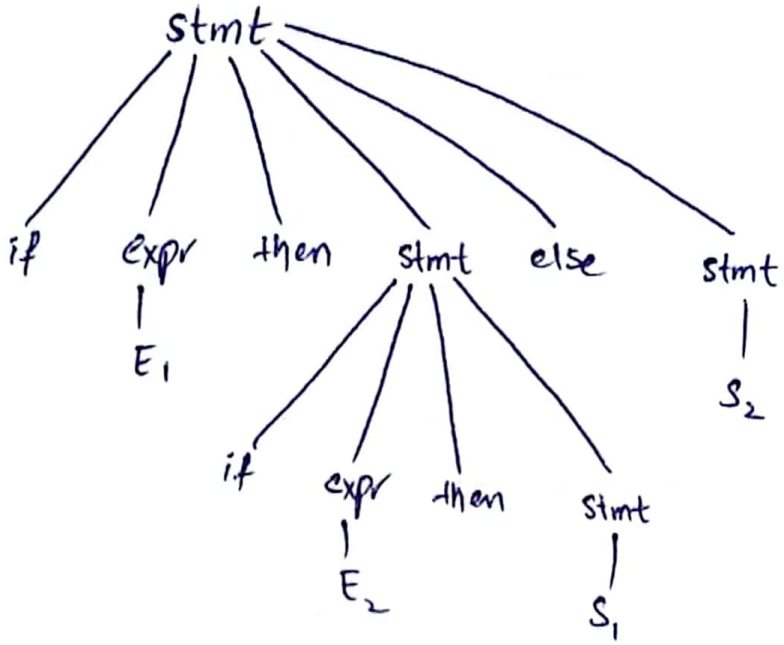
parse tree 1:



parse tree 2: if E_1 then if E_2 then S_1 else S_2



parse Tree 2:



"Ambigous Grammer"

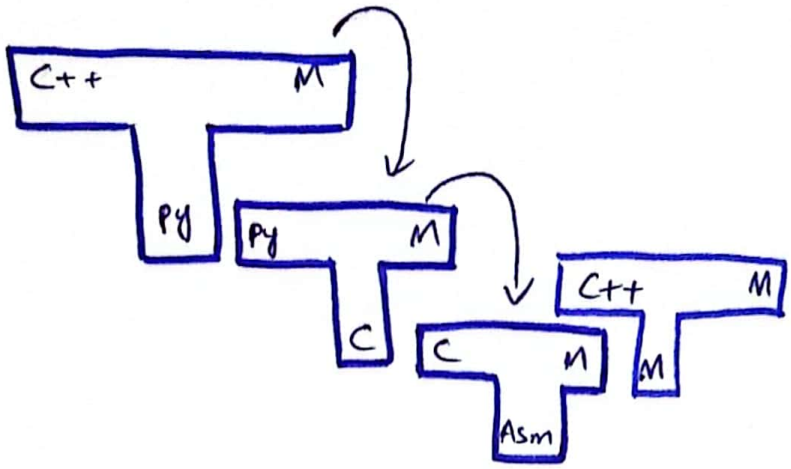
Types of parsing

→ Top Down construct parse tree from root to leaf

→ Bottom up " leaf to root or ~ Top Down

Bootstraping :

Language in which compiler is written is not understandable by target machine, we need to write another compiler which translates its instructions in target machine



Left & Right Recursion:

(1) Left Recursion: Nonterminal before '→' is present after '→' in 1st position.

For example:

$$A \rightarrow A\alpha / \beta$$

(2) Right Recursion: opposite of left recursion

$$A \rightarrow \alpha A / \beta$$

→ Left Recursion need to remove from CFG

Why not Righ recusion?

Consider two scenarioes of recursion in programming

S1:	S2:

A() {
 A();
 α
 }
	B() {
	 B
	 B();
	 }

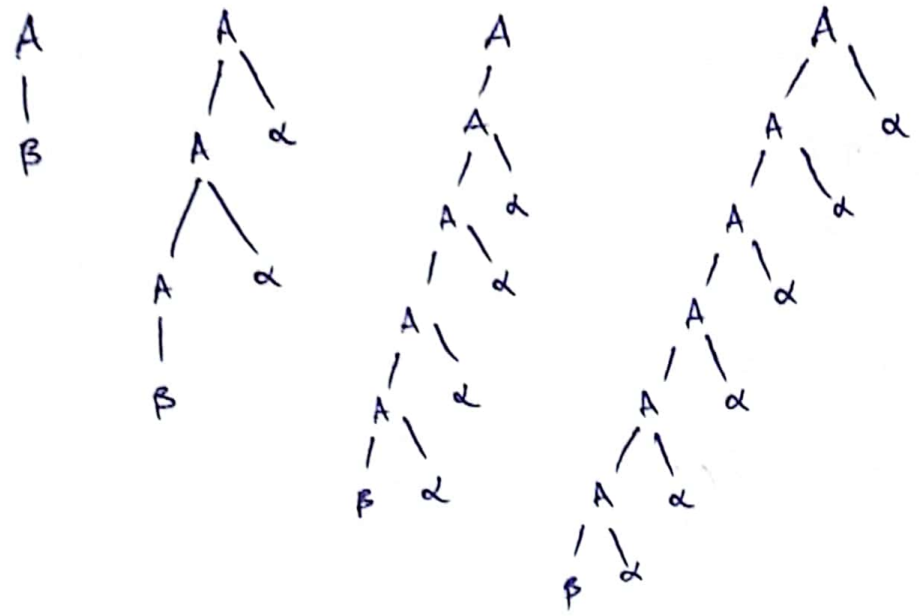
$1 will have issue that no pre-condition will be checked due to left recursion while other case is perfect and has no major issue.

That's why left recursion must be removed from CFG

Example:

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

parse Trees:



possible values = $\{ \beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha \cdots, \}$

$$RE = \beta\alpha^*$$

Rewriting CFG:

$$A \rightarrow \beta A'$$

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

we can map this example on all other cases by just recongnizing α, β, A & A'

Example:

(1) Remove left recursion from given CFG

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

Sol:

$$\underset{A}{E} \rightarrow \underset{A}{E} \underset{\alpha}{+ T} \mid \underset{\beta}{T}$$

$E = A$, $+T = \alpha$, $T = \beta$

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

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

(2) Remove LR from CFG

$$\underset{A}{S} \rightarrow \underset{A}{S} \underset{\alpha}{0S1S} \mid \underset{\beta}{01}$$

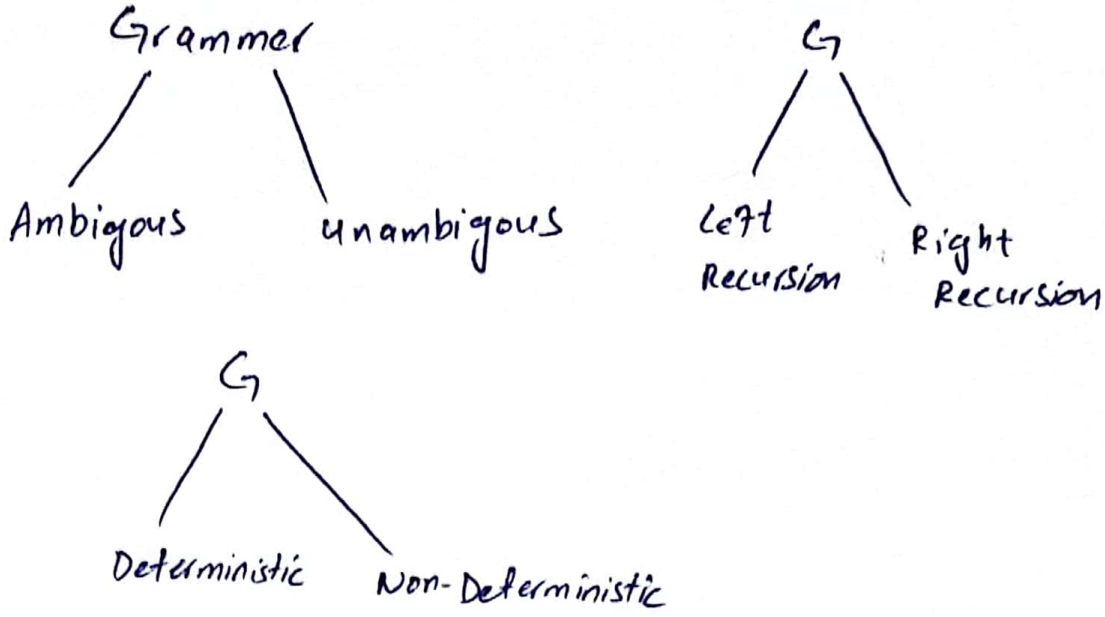
$$S \rightarrow 01 \mid S'$$
$$S' \rightarrow 01S1SS' \mid \varepsilon$$

(3) Identify LR exist?

a) $S \rightarrow (L) \mid x$

b) $S \rightarrow SL \mid x$

c) $L \rightarrow L, S \mid S$



Deterministic Grammer: (Left factoring)

- If multiple opportunities not available for a single input in CFG, it is deterministic otherwise non-deterministic

Example:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

Lets say compiler got $\alpha\beta_3$, it will ask for production rule related to $\alpha\beta_3$. CFG will return $\alpha\beta_1$ then $\alpha\beta_2$ next time. The main problem is not that we have to traverse the rules, but we have to compare α' everytime when we got rule from CFG which is redundant.
we need to put determinism in CFG by taking common as a separate rule:

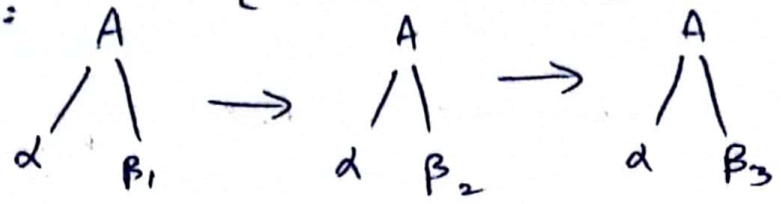
$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

In this case, we introduced new rule A' by taking 'α' as common.

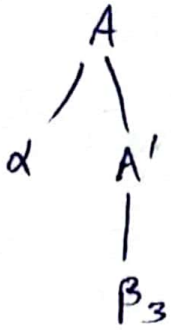
Comparison of parse tree for both cases:

[Non-Deterministic]

Case#1:



Case#02: [Deterministic]



Example 2:

Convert CFG to deterministic one:

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

Sol

Common = 'iEtS' => S'

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

$$S' \rightarrow \varepsilon \mid eS$$

Example 3:

$$S \rightarrow aSSbS \mid aSaSb \mid abb \mid b$$

Sol

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SSbS \mid SaSb \mid bb$$

still require determinism

$$S \rightarrow aS' \mid b$$

$$S' \rightarrow SS'' \mid bb$$

$$S'' \rightarrow SbS \mid aSb$$

Why searching is not used as compare to parsing?

Searching a string from CFG requires finding the appropriate rule each time. As # of production rules increases, # of child increases. What if string not exist in CFG rules, wastage of time.

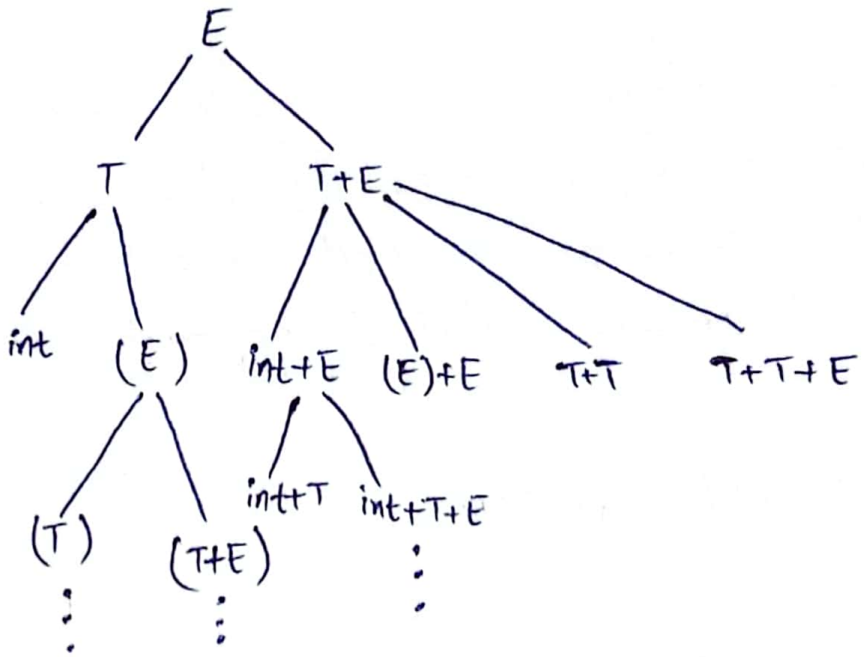
Example:

Let's use Breadth first search **(BFS)** for searching:

CFG:

$E \rightarrow T \mid T+E$

$T \rightarrow int \mid (E)$



BFS = Level wise from left to right traversal

Lets say input string is 'int+int', we need to find all paths from tree till int+int found (if exist according to rule).

That's why searching via BFS is not used.

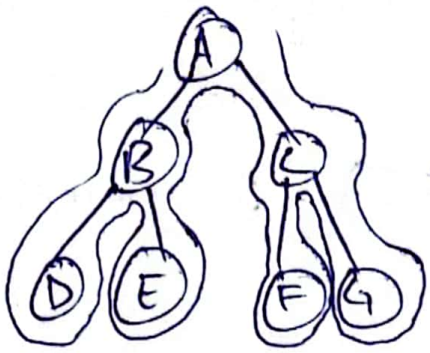
Branching factor = max # of child allowed for tree

Left most BFS:

only expand left most child and compare single character with input string whether to expand more or pick another node.

Still require lot of search to reach valid sequence.

DFS



It is in favour of leftmost but it can cause problem in case of graphs and leads to infinite search.

We need algorithm which guide exact rules which need to apply to get final result.

predictive parsing is used to get exact rules and provides efficient approach towards parsing.

It uses LL(1) parser.

LL (1)

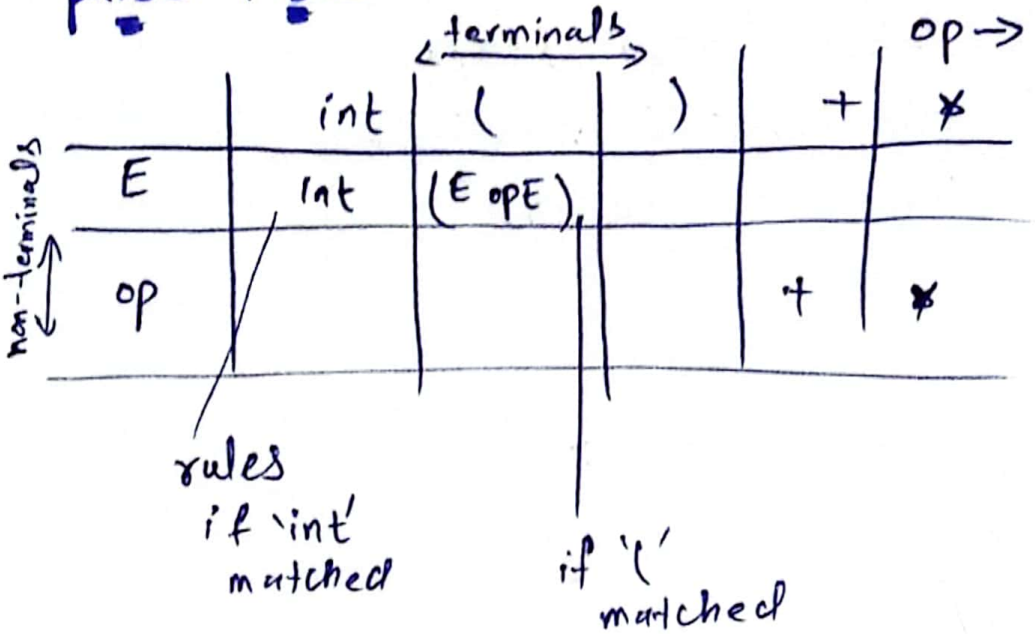
- left to right
- left most derivation
- one token of lookahead

CFG:

$E \rightarrow int \mid (E\ op\ E)$

$op \rightarrow + \mid *$

parse Table



Guidline:

- pick on tokem and check corresponding rule from parse table, if not found → discard
- if '$' matched at end, accept
- remove matched character one by one

Example:

(int + (int * int))

put '$' at end of rules & input string

Rules	string
E $	(int+(int * int)) $
'(' (E op E) $	int + (int * int)) $
'int' int op E) $	+ (int * int)) $
'+' + E) $	(int * int)) $
'(' (E op E)) $	int * int)) $
'int' int op E)) $	int * int)) $
'*' op E)) $	* int)) $
* E)) $	* int)) $
E)) $	int)) $

'int' int)) $ int)) $

"matched"

How to generate "parse table"?

put terminals on top row and non-terminals on left most column.

Issue in LL(1) parser is that we are looking only first token, what if 1st token is itself non-terminal which leads toward tracing until terminal is found.

For example:

stmt → if expr then stmt (1)
| while expr do stmt (2)
| expr; (3)
| Term (4)

expr → Term → id | zero? term (5)
| not expr (6) | ++id (7) | --id (8)

term → id (9) | constant (10)

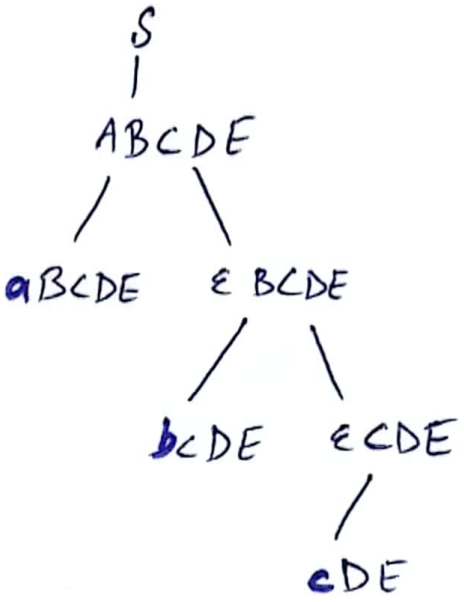
Rule (3) and (4) have 'expr' & 'term' as 1st token which are not terminal.

For creating 'parse table', these entries can be easily applied using **first()** and **follow()** set.

first() set:

- place all the first terminals in a set

Rules	First set	follow()
A → a/ε	{a, ε}	{b, c}
S → ABCDE	{a, b, c}	{$}
B → b/ε	{b, ε}	{c}
C → c	{c}	{d, e, $}
D → d/ε	{d, ε}	{e, $}
E → e/ε	{e, ε}	{$}



Example:

Rules	first() set	follow()
S → Bb / Cd	{a, b, c, d}	{$}
B → aB / ε	{a, ε}	{b}
C → cC / ε	{c, ε}	{d}

For follow set:

→ for starting rule, it will be always {$}

→ if no follower, then put all the follow set elements of leftmost (before arrow)

For first(): put all the first terminals in a separate se

Example 3:

Find first() and follow() set of given Grammer:

1) $E \rightarrow TE'$

2) $E' \rightarrow +TE' / \varepsilon$

3) $T \rightarrow FT'$

4) $T' \rightarrow *FT' / \varepsilon$

5) $F \rightarrow id \mid (E)$

→ First():

1) {id, (}

2) {+, ε}

3) {id, (}

4) {*, ε}

5) {id, (}

→ Follow():

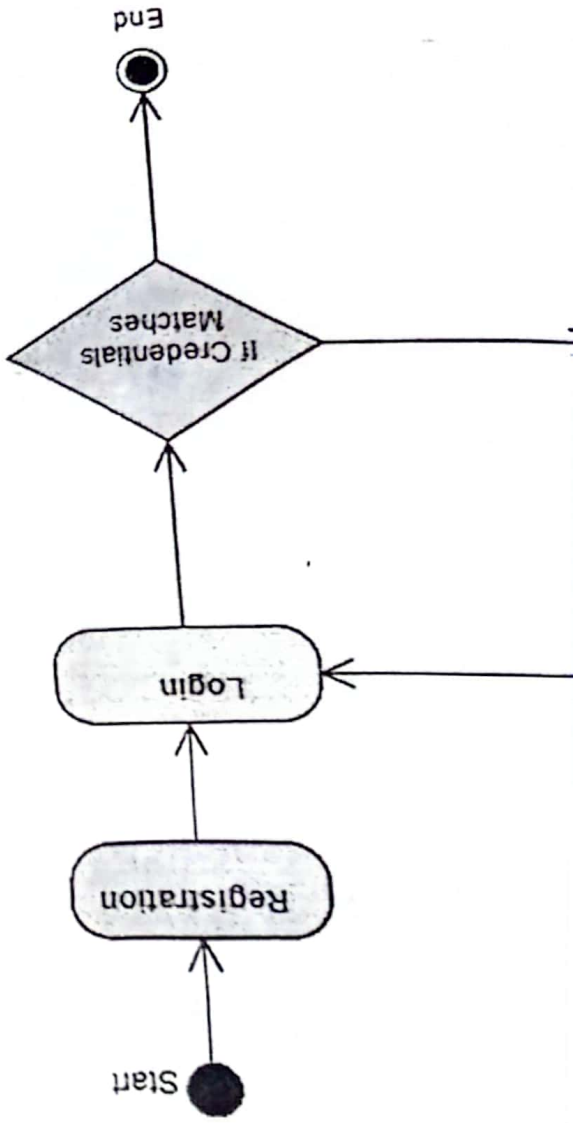
1) {$,)}

2) {$,)}

3) {+, $,)}

4) {+, $,)}

5) {*, +, $,)}



	first()	follow()
① S → ABC | ghi | jkl	{a, g, j, b, c}	{$}
A → a | b | c	{a, b, c}	{b}
B → b	{b}	{c}
D → d	{d}	{ }
C → c	{c}	{b}

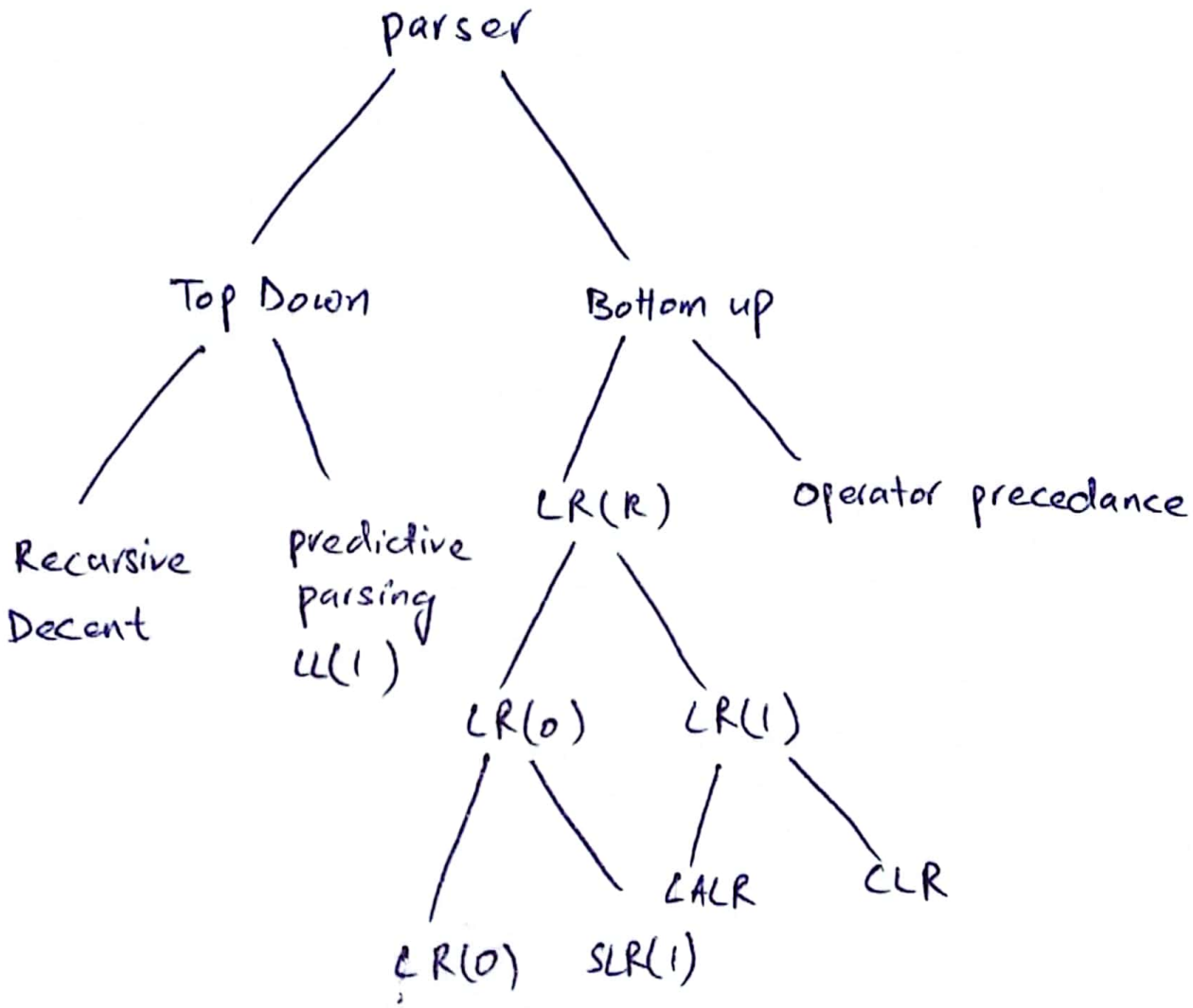
②

S → ABC {$}

A → DEF {$}

B, C, D, E, F → ε {$}

Types of parsing:



How to check Grammer is LL(1)?

1) If epsilon 'ε' not in a rule, find intersection of first() set, if equal to ϕ, LL(1) otherwise not

2) if epsilon 'ε' exist in rule, also find follow() set of that non-terminal and take intersection with first() of others

Example:

→ Case 1:

$$S \to \alpha 1 \mid \alpha 2 \mid \alpha 3$$

$$first(\alpha 1) \cap first(\alpha_2) \cap first(\alpha_3) = \phi$$

→ Case 2:

$$S \to \alpha 1 \mid \alpha 2 \mid \alpha 3 \mid \varepsilon$$

$$first(\alpha 1) \cap first(\alpha 2) \cap first(\alpha_3) \cap follow(S) = \phi$$

$S \to aSa \mid bS \mid c$

is LL(1) or not?

$first(aSa) \cap first(bS) \cap first(c)$

$= \phi$

Yes, gts LL(1) Grammer

$S \to iCtSS_1 \mid a$

$S_1 \to eS \mid \varepsilon$

$C \to b$

① $first(iCtSS_i) \cap first(a)$

$i \cap a = \phi$

② $first(eS) \cap follow(S_1)$

$e \cap \$e \neq 0$

gts not LL(1) parsing Grammer!

Apply LL(1) parsing:

① $S \rightarrow aABb$

② $A \rightarrow c \mid \varepsilon$

③ $B \rightarrow d \mid \varepsilon$

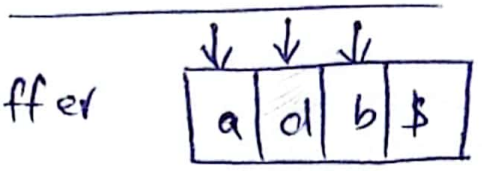
	First()	follow()
1)	{a}	{$}
2)	{c, ε}	{d, b}
3)	{d, ε}	{b}

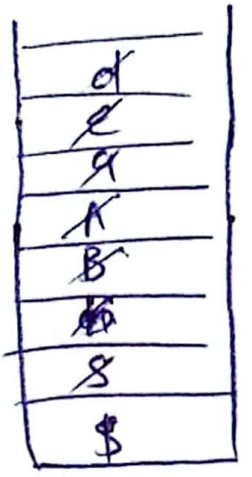
parsing Table:

	a	b	c	d	$
S	S→aABb				
A		A→ε	A→c	A→ε	
B		B→ε		B→d	

if 'ε' in rule, then put follow() set entries the rule

For example: in 2nd rule A→c|ε, epsilon exist, so check follow set which are {d, b}. put the rule A→ε in 'b' and 'd' column in parsing table

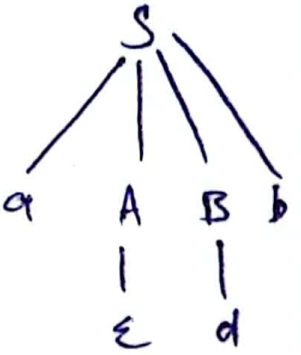




Stack

if $ symbol at the end in the stack, accept the string.

parse tree will be generated side by side upon pushing in stack



Example:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

Is LL(1) parsing possible?

Try to create parse table, if more than one entry in a table ~~ex~~ box exist, then LL(1) parsing not possible.

As in the given example, there is no 'ε', so just find first() set.

	first()
$E \rightarrow E+T \mid T$	{id}
$T \rightarrow T*F \mid F$	{id}
$F \rightarrow id$	{id}

Since Left Recursion exist, LL(1) not exist.

(2)

	first()	follow()
$A \rightarrow Bb \mid Cd$	{a, b, c, d}	{$}
$B \rightarrow aB \mid \varepsilon$	{a, ε}	{b}
$C \rightarrow cC \mid \varepsilon$	{c, ε}	{d}

Check intersection of first() if no 'ε' otherwise also take intersection with follow() excluding 'ε'

For previous example

Rule 1 $\{a, b\} \cap \{c, d\} = \phi$ because no 'ε', just pick first() set

Rule 2 $\{a\} \cap \{b\} = \phi$ because 'ε' exist, pick first() & follow

Rule 3 $\{c\} \cap \{d\} = \phi$ same as rule 2

So, LL(1) possible!

Operator precedance parser:

- Bottom up parser
- CFG must have operator b/w non-terminals (operator grammer)

Example:

① $E \rightarrow E+E \mid E*E \mid id$

It's operator grammer because non-terminals are separated by +, * operator

② $E \rightarrow E \ op \ E \mid id$

$op \rightarrow + \mid - \mid *$

Its not operator grammer due to 'op' non-terminal between two non-terminals.

Example:

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

Apply operator precedance parsing. (operator relational table)

①

	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	accept

precedance of 'id' is greater than '$'

identifier has highest precedance

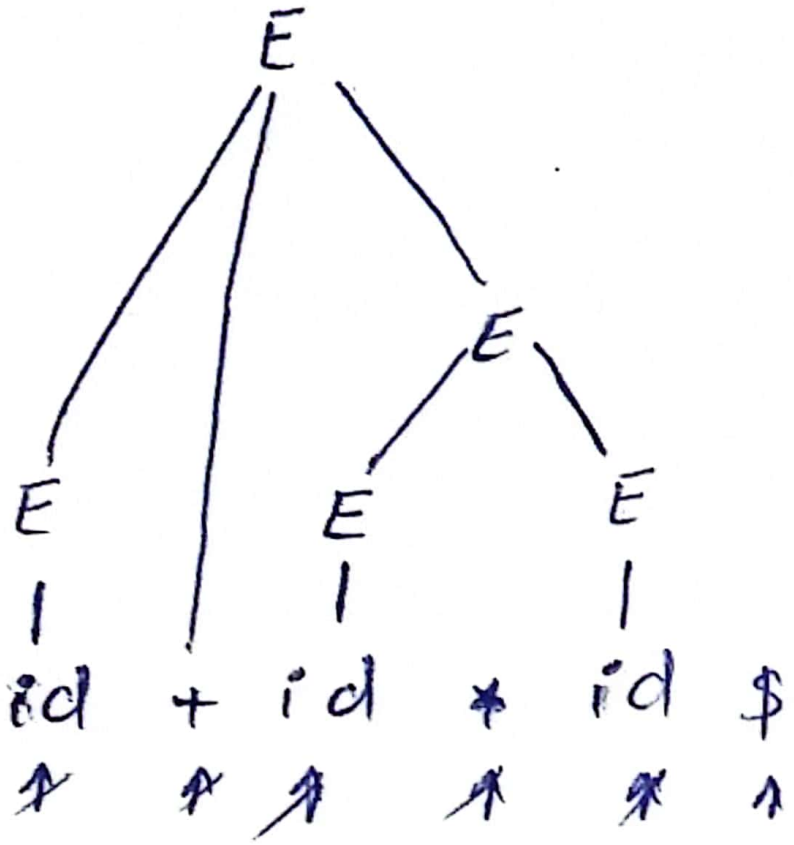
$id > * \mid \div > + \mid -$

precedance of leftmost '*ply' is > than rightmost '*ply

② Create stack and

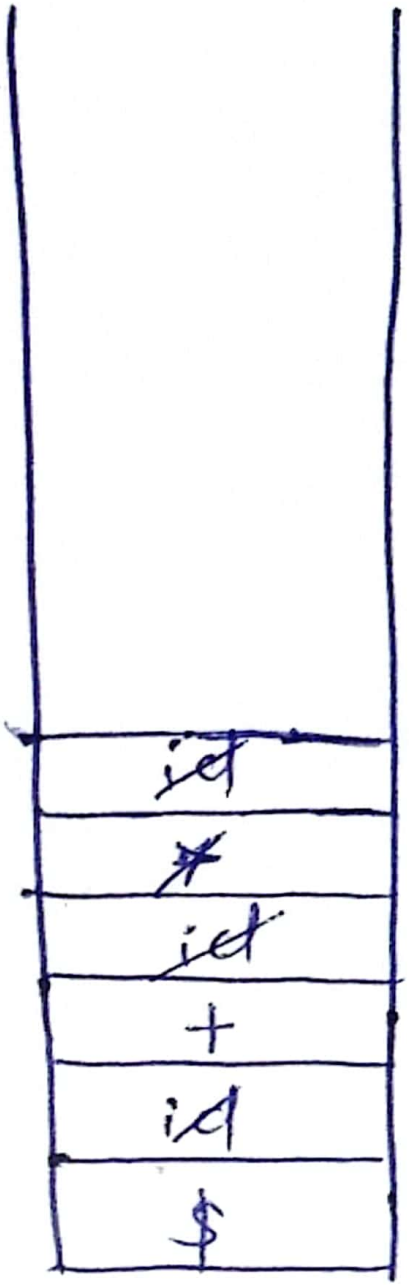
push = element of current pointer has > precedance than stack element

POP = element in the stack has high precedance



→ Compare element in the pointer with element in the stack & decide to push or pop

Stack

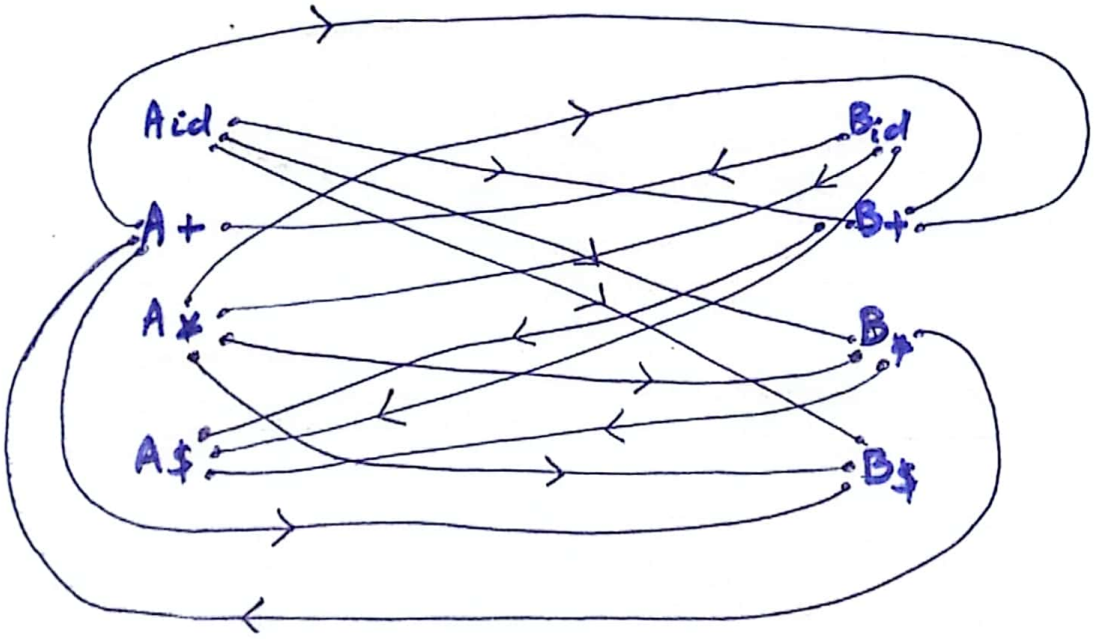


In operator precedance parser, for 'n' terminals table size will be nxn which will consume more space if grammer size is large.

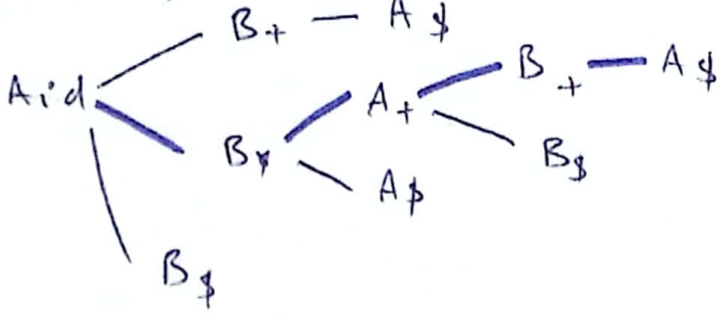
we need some mechanism to reduce table size.

This parser also accepts ambigous grammer.

A \ B	id	+	*	$
id	-	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	Accept

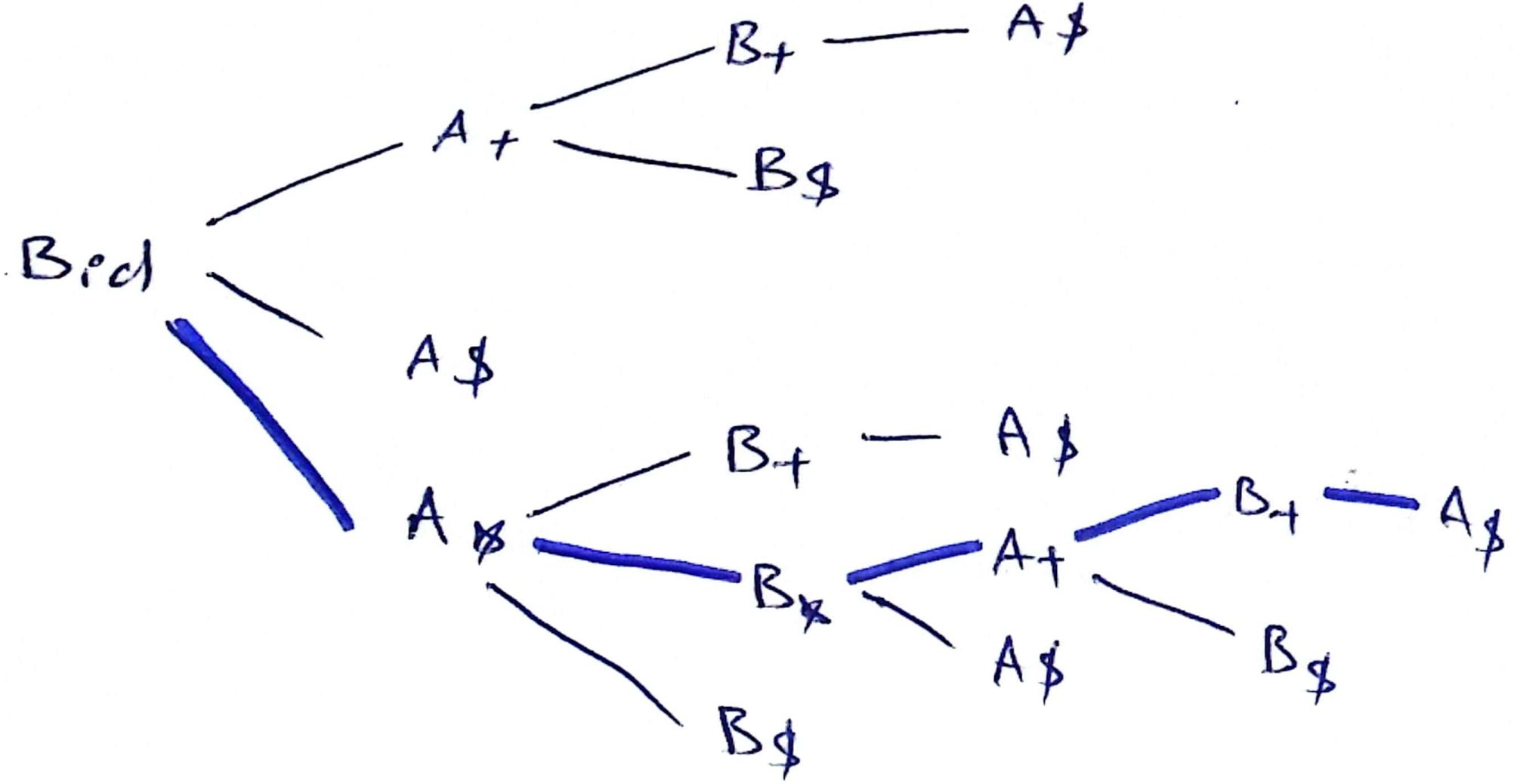


Find longest path



put Aid → B* → A+ → B+ → A$ is missing 3 terminals A*, B$, Bid

Now!


B+
A$
A+
B$
Bid
A$
B+
A$
A*
B+
A$
B*
A+
B$
A$
B$

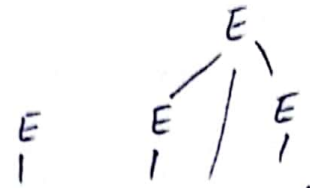
$A_{id} \rightarrow B_{*} \rightarrow A_{+} \rightarrow B_{+} \rightarrow A_{\$}$

$B_{id} \rightarrow A_{*} \rightarrow B_{*} \rightarrow A_{+} \rightarrow B_{+} \rightarrow A_{\$}$

	id	*	+	$
A	4	4	2	0
B	5	3	1	0

"operator functional Table"

Example =



B: id + id * id $

1) $A_{\$}$ > or < Bid
   0 < 5 "push"

2) Aid > or < B+
   4 > 1 "pop"

3) A$ <or> B+
   0 < 1 "push"

4) A+ <|> Bid
   2 < 5 "push"

5) Aid <|> B*
   4 > 3 "pop"

6) A+ <|> B*
   2 < 3 "push"

7) A* <|> Bid
   4 < 5 "push"

8) Aid <|> B$
   4 > 0 "pop"

9) A* <|> B$
   4 > 0 "pop"

10) A+ <|> B$
    2 > 0 "pop"

11) A$ = B$
    "accept"

precedence & Associativity in CFG:

Some times, CFG voilates precedance or associativity property which leads toward invalid results.

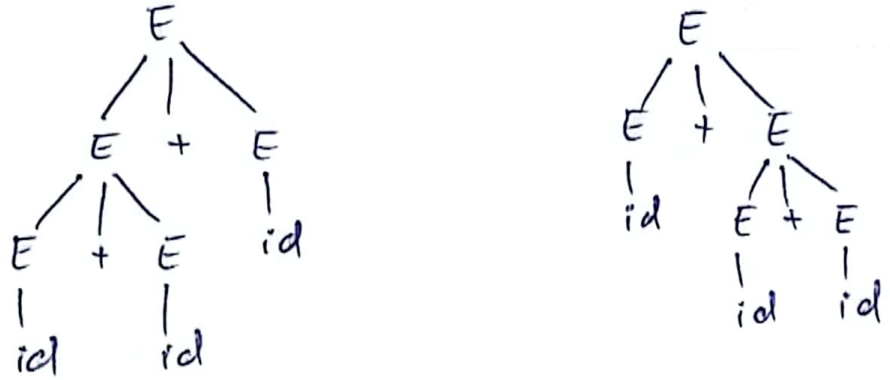
For example:

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

Let say we have input string

$$id + id + id$$

parse Tree:



Ambiguity exist in this case, one tree favours right associativity (id+id) +id and other one left associativity id + (id+id).

To make right associative, we can update grammar by making it right recursive. In the same way, for left associative, make grammer left recursive Higher/deeper the level of operator, highest the precedance will be.

As in this case, we need left associative, convert grammer to left recursive one.

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

b → b and b
| b or b
| not b
| true
| false

b → b and c | c

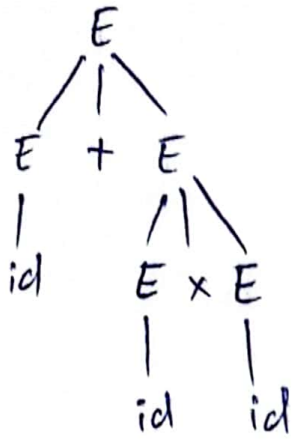
c → c or d | d

d → true | false | not d

Example 2: (precedance violation)

$E \rightarrow E+E \mid E \times E \mid id$

input string = id+idxid



" right associative"

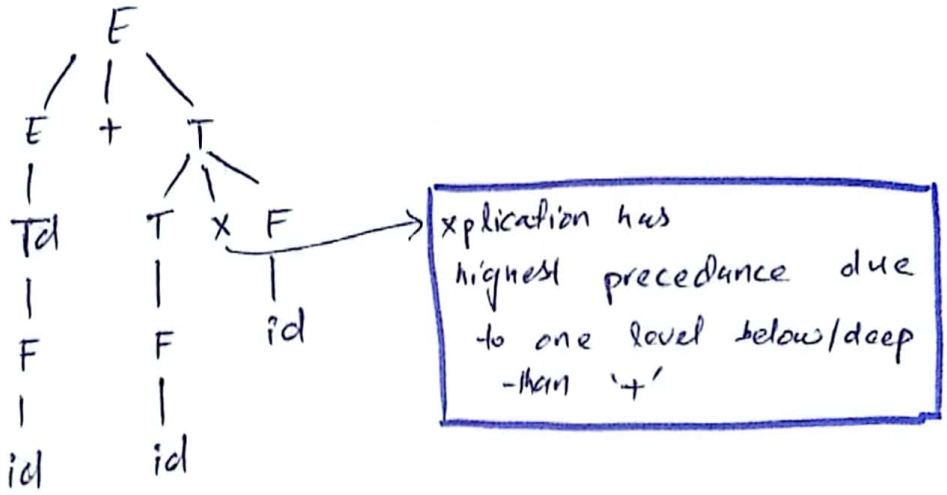
Making grammer left recursive

$E \rightarrow E+T \mid T$

$T \rightarrow T \times F \mid F$

$F \rightarrow id$

Again making parse tree on new grammar



Example 3:

Determine precedance of operators in given grammer

$E \rightarrow E+T \mid T$

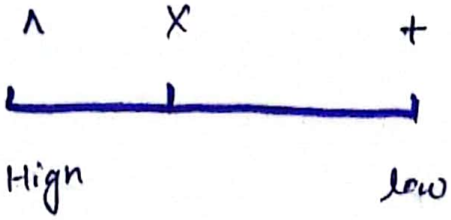
$T \rightarrow T \times F \mid F$

$F \rightarrow G \wedge F \mid G$

$G \rightarrow id$

Try to parse the target operator from start symbol, operator having longest depth will have highest precedance

precedance



Example 4:

Consider given ambigous grammer:

bExp → bExp AND bExp | bExp OR bExp | NOT bExp
| True | False

precedance of operators NOT > AND > OR

Solution

bExp → bExp OR bExp1 | bExp1

bExp1 → bExp1 AND bExp2 | bExp2

bExp2 → NOT bExp2 | True | False

Example 5:

R → R+R | R.R | R* | a | b | c

precedance = * > . (and) > + (or)
(left to right associativity)

Solution

R → R+R1 | R1

R1 → R1.R2 | R2

R2 → R2* | a | b | c

Example 6:

S → TxP

T → U | TxU

P → Q+P | Q

Q → id

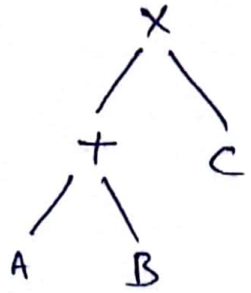
U → id

In this example,
'+' has highest precedance than 'x'

+ = right associative due to right recursion

x = left associative

Which of the following expression is presented by parse tree?



→a) (A+B) x C

b) A + x BC

c) A+Bx C

d) Ax C+B

Example:

BE → BE and BF | BE or BF | (BE) | B

BF → true | false

* Rewrite CFG by applying rule
() > and > or => precedance

Solution:

BE -> BE or BF | B1
B1 -> B1 and B2 | B2
B2 -> (B2) | BF
BF -> true | false

Example:

E → E+E | E-E | ExE | E/E | id | num

+ has highest precedance while other have same. right to left associativity

Sol:

E -> E1-E | E1xE | E1/E | E1
E1 -> E2+E1 | E2
E2 -> id | num

① $E \rightarrow E<E \mid E>E \mid E==E \mid E!=E \mid num$

need precedance of equality & inequality is > than other

Sol:

$E \rightarrow E<E \mid E>E \mid E1$

$E1 \rightarrow E1==E1 \mid E1!=E \mid E2$

$E2 \rightarrow num$

② $BE \rightarrow BE \text{ and } BE \mid BE \text{ or } BE \mid \text{not}(BE) \mid \text{true} \mid \text{false}$

not > and > or => precedance

a) $BE \rightarrow BE \text{ or } BE \Rightarrow BE1$

$BE1 \rightarrow BE1 \text{ and } BE1 \mid BE2$

$BE2 \rightarrow \text{not } BE2 \mid BE3$

$BE3 \rightarrow \text{true} \mid \text{false} \mid (BE)$

b) make '&' and '||' right associative

$BE \rightarrow BE \text{ or } BE1 \mid BE1$

$BE1 \rightarrow BE1 \text{ and } BE2 \mid BE2$

$BE2 \rightarrow \text{not } BE2 \mid BE3$

$BE3 \rightarrow \text{true} \mid \text{false} \mid (BE)$

③ $S \rightarrow S\&S \mid S=S \mid S\#S \mid id \mid num$

&, # > other => precedance (left assoc)

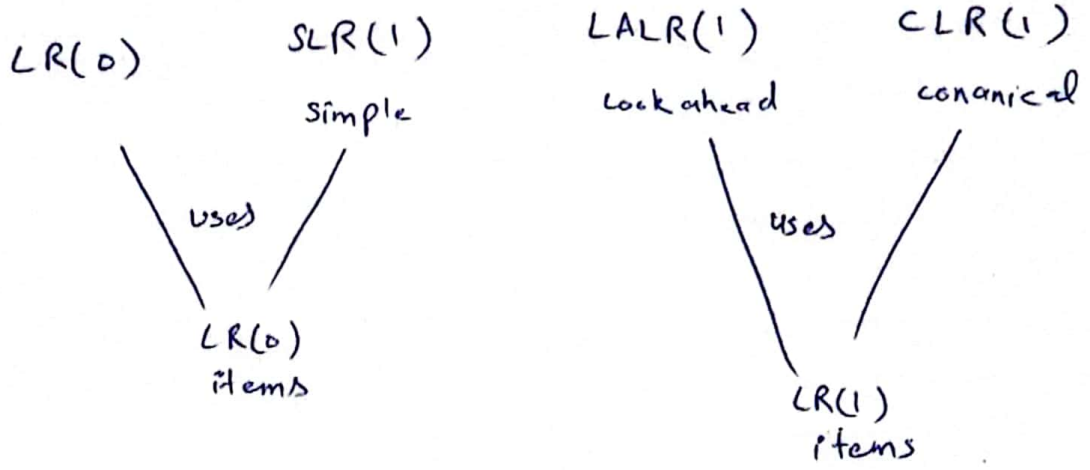
Sol

$S \rightarrow S=S1 \mid S1$

$S1 \rightarrow S1 \& S2 \mid S1 \#S2 \mid S2$

$S2 \rightarrow id \mid num$

Bottom up parser:



LR(0) items:

- Used in LR(0) & SLR(1) with little modification

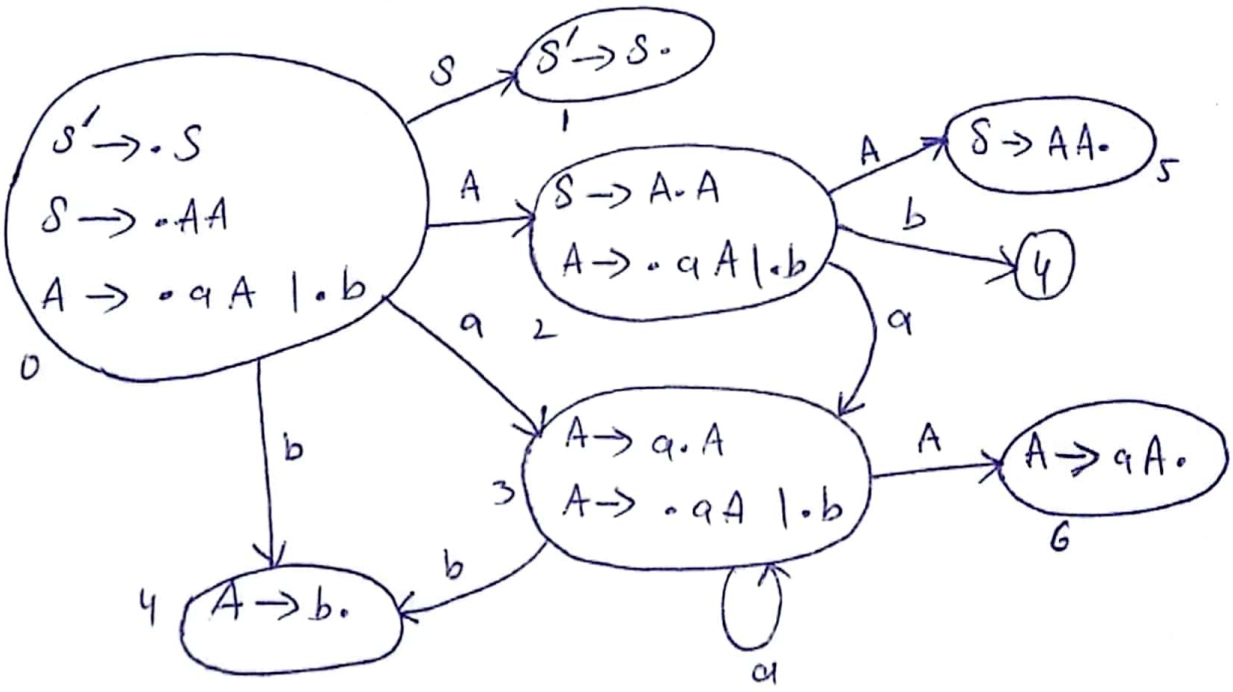
Example:

$S' \rightarrow S$ rule 0

$S \rightarrow AA$ rule 1

$A \rightarrow aA \mid b$ rule 2 & 3

put dot '.' at start of each right side symbol
shift '.' one place. If non-terminal exist after dot '.', expand it
Continue till dot shift at the end



Canonical collection of LR(0) items

States	Action			Go to	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4					
5					
6					
7					

Till this point, LR(0) & SLR(1) are Same

Now LR(0):

States	Action			Go to	
	a	b	$	S	A
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

$A \rightarrow b.$

State no: 4
rule no: 3 or r_3

$A \rightarrow aA$

State no: 6
rule no: 2 or r_2

$S \rightarrow AA$

State no: 5
rule no: r_1

put rule of each state in corresponding terminal row

Lets parse "aabb$"

aabb$
↑ ↑ ↑

0	a	3	~~a~~	~~3~~	~~b~~	~~4~~	~~A~~	~~6~~	A	6	

for 1st character 'a', state = 0,
Action = S3 → push 'a' & '3' on stack
(There must be a number on top of stack)
If rule is encountered, do 4 tasks

① 2 times pop
② push LMS
③ Tree
④ put Goto num

If r_n is encountered

1) pop 2 ~~demands~~ times #1 of element/characters on right side of rule

2) push corresponding rule left most side non-terminal

$A \rightarrow b$

pick number before 'A' which is 3 in this case,

push element having ~~Action~~ state = 3, q_0 to = A which is 6, so push '6'

pick rule of state 6

$A \rightarrow aA$

SLR(1):

- find follow() set of Grammer

$S' \rightarrow S$ {$}

$S \rightarrow AA$ {a, b, $}

$A \rightarrow aA \mid b$

State	Action			S	A
	a	b	$		
0	S_3	S_4		1	2
1			Accept		
2	S_3	S_4			5
3	S_3	S_4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

Remaining part of parsing is same as LR(0).

we can see LR(0) has more entries than SLR(1). The less the entries in table, more chance of detecting error.

(1) $A \rightarrow b$
state = 4
rule = 3
put 'r3' under follow() set terminals of A
{a, b, $}
It means 'r3' will come under a, b, $ because these are follower of 'A'
Same for other

(2) $A \rightarrow aA$
$S = 6, r = 2$
follow = {a, b, $}

(3) $S \rightarrow AA$
$r = 1,$
$s = 5$
follow = {$}

Example:

Appig LR(0) & SLR(1) parsing:

$S \rightarrow dA \mid aB$

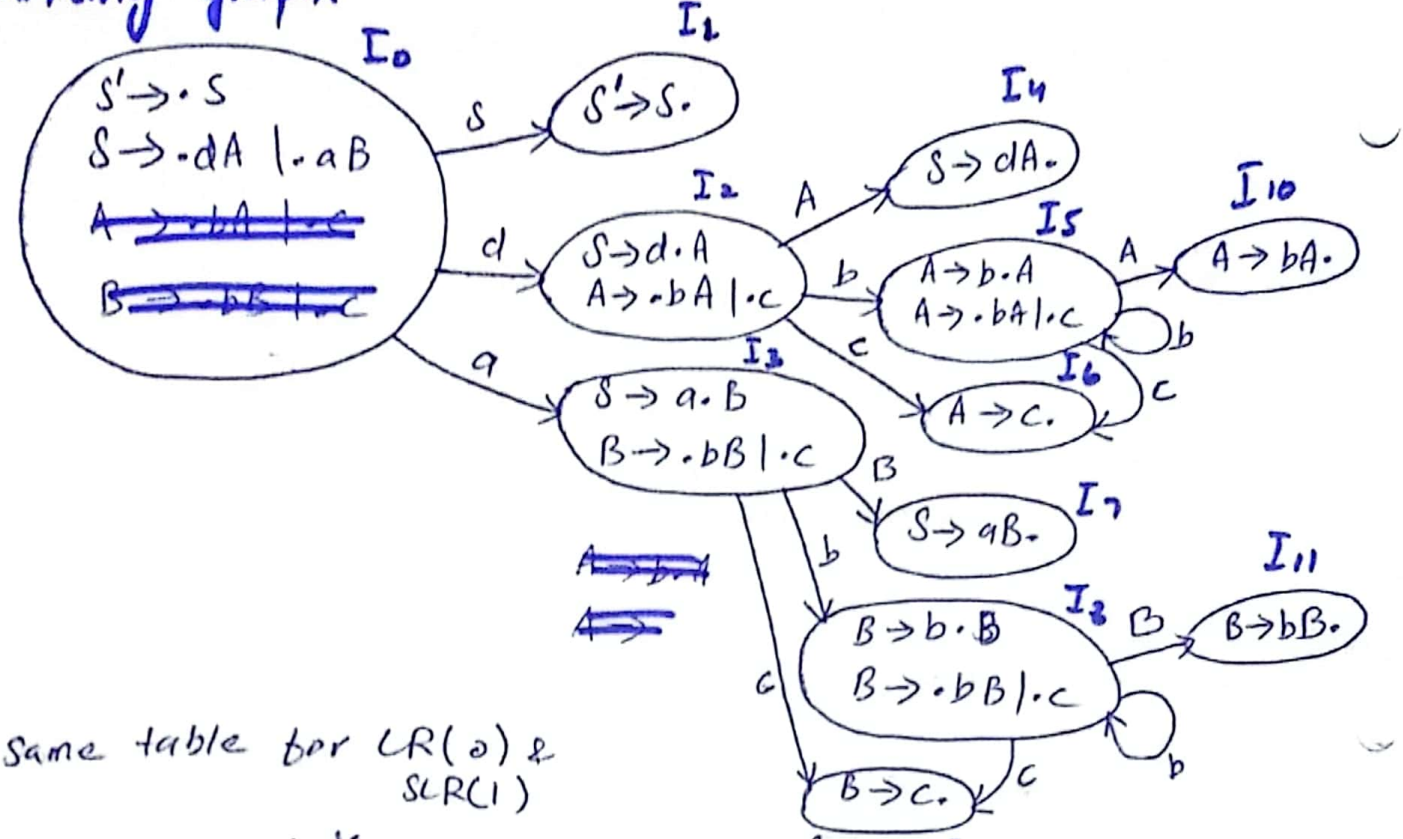
$A \rightarrow bA \mid c$

$B \rightarrow bB \mid c$

Solution: adding additional state

$S' \rightarrow S$

Shifting graph:



Same table for LR(0) & SLR(1)

States	Actions a	b	c	d	$	Goto S	A	B
0	S_3			S_2		P		
1								
2		S_5	S_6				4	
3		S_2	S_9					7
4								
5		S_5	S_6				10	
6								
7								
8		S_8	S_9					11
9								
10								
11								

LR(0):

	rule # & state #	follow()
$S' \to S$		
$S \to dA$	— (1) r1, S4	{$}
$\mid qB$	— (2) r2, S7	
$A \to bA$	— (3) r3, S10	{$}
$\mid c$	— (4) r4, S6	
$B \to bB$	— (5) r5, S11	{$}
$\mid c$	— (6) r6, S9	

S_n = state #

r_n = rule #

putting rules in table:

State	Action a	b	c	d	$	S	A	B
0	S_3			S_2		1		
1								
2		S_5	S_6				4	
3		S_8	S_9					7
4	r1	r1	r1	r1	r1			
5		S_5	S_6				10	
6	r4	r4	r4	r4	r4			
7	r2	r2	r2	r2	r2			
8		S_8	S_9					11
9	r6	r6	r6	r6	r6			
10	r3	r3	r3	r3	r3			
11	r5	r5	r5	r5	r5			

Just remove box entries for SLR(1)

main difference in LR(0) and SLR(1) is table generation for each rules

In LR(0), we fill complete row of corresponding state with rule #, but in SLR(1), just put rule # under those 'follow set' and 'Action' common terminals

LALR(1) & CLR(1)

- Similar to LR(0) & SLR(1) but add "look ahead" as well
- Lookahead is like follow set
  For starting state, it will always be '$'

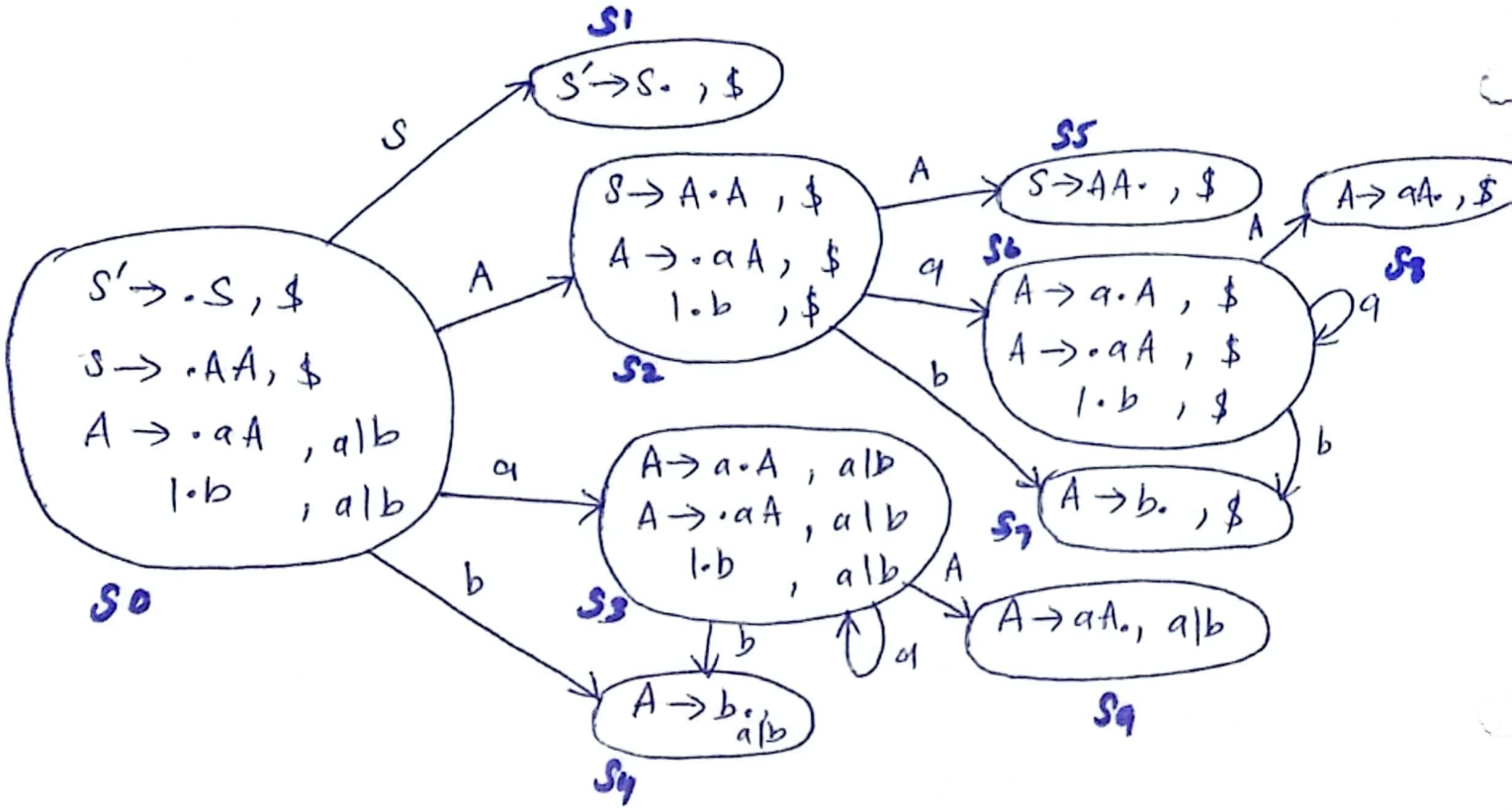
Example:

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

Sol: we always add additional state $S' \rightarrow S$, in all bottom up parsers.

	Lookahead
$S' \rightarrow S$	$
$S \rightarrow AA$	$
$A \rightarrow aA$	a|b

→ copy look ahead of non-terminal state as it is



LALR(1)

State	action a	b	$	Go-to S	A
0	S_3	S_4		1	2
1					
2	S_6	S_7			5
3	S_3	S_4			9
4	r3	r3			
5			r1		
6	S_6	S_7			8
7			r3		
8			r2		
9	r2	r2			

		rule # 'r$_n$', state #, S_n, look-ahead
S → A A		S_5, r1	{$}
A → aA		S_9, S_8, r2	{$}, {a|b}
| b		S_7, S_4, r3	{a|b}, {$}

CLR(1)

Looking at transition diagram, below states are similar

S_8, S_9 => S_{89} (new state)

S_3, S_6 => S_{36}

S_4, S_7 => S_{47}

Combine these states in the table computed via LALR(1)

Table must be in LALR(1) to find CLR(1)

CLR(1) reduces # of states but increases table entries

LALR(1) & CLR(1) uses LR(1) items

State	Action			Go to	
	a	b	$	S	A
0	S_{36}	S_{47}		1	2
1					
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

problem in LALR(1) & CLR(1)

if same lookahead exist in a state, where multiple rules exist then there is a conflict

For example:

$A \rightarrow ab$

$B \rightarrow cd$

$A \rightarrow ab.$, $

$B \rightarrow cd.$, $

So

Table

State	Action				
	a	b	c	d	$
0					r1/r2

Problem/conflict

Example:

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

$A \rightarrow d$

$B \rightarrow d$

Create LALR(1) & SLR(1) parser table.

Sol:

$S' \rightarrow S$

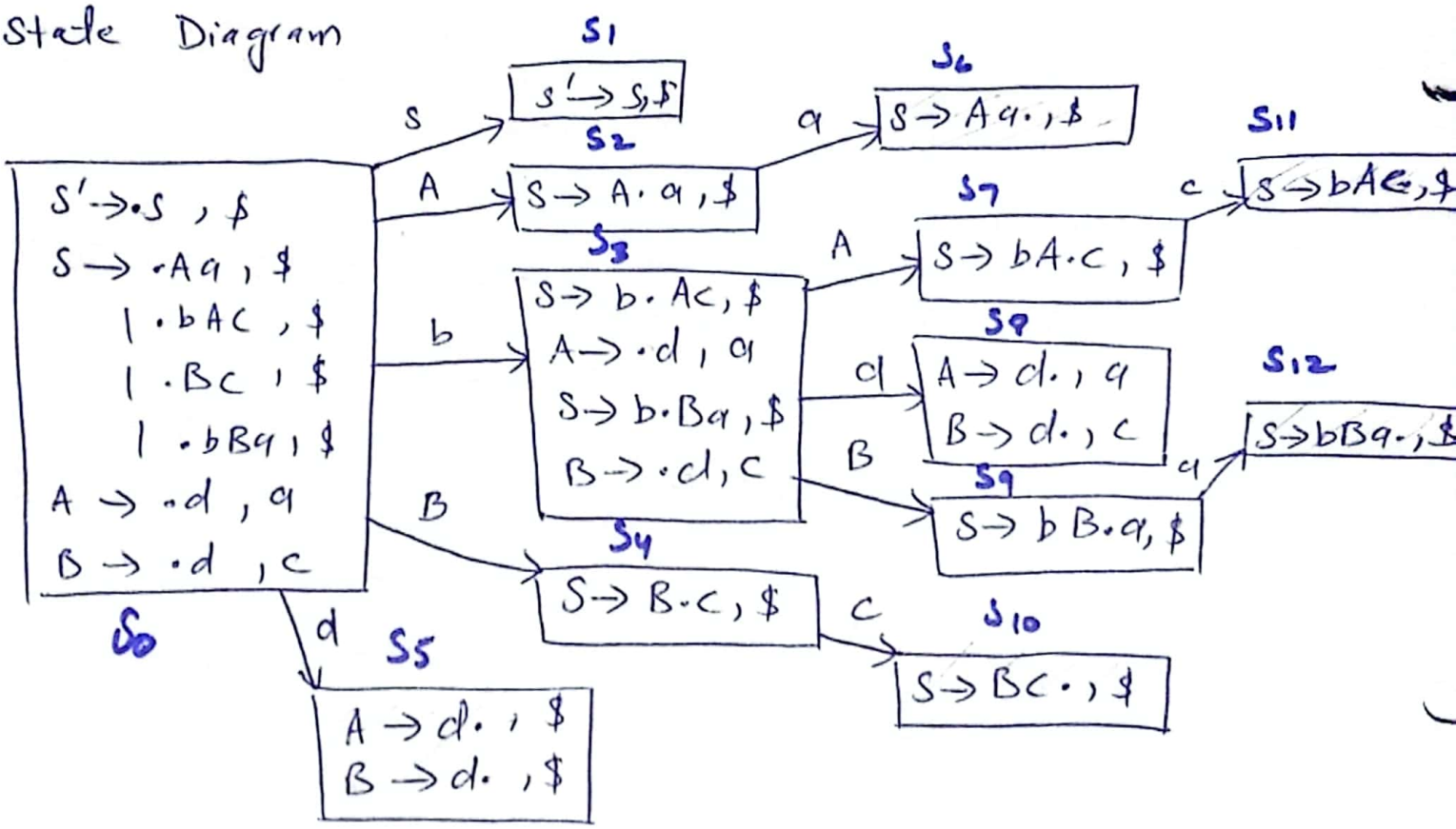
$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

rule#,	state #,	lookahead
//		$
r1,	s_6,	$
r2,	s_{11},	$
r3,	s_{10},	$
r4,	s_{12},	$
r5,	s_5,	s_8, a
r6,	s_5,	s_2, c

State Diagram



State	Action							
	a	b	c	d	$	S	A	B
0		S_3		S_5		S_1	S_2	S_4
1								
2	S_6							
3				S_7			S_8	S_9
4			S_{10}					
5	r_5		r_6					
6					r_1			
7			S_{11}					
8	r_5		r_6					
9	S_{12}							
10					r_3			
11					r_2			
12					r_4			

If CLR(1) is not possible due to conflict, LALR(1) is also not possible.

Recursive Descend parser

- top down parser
- Implement separate method/routine/function for each non-terminal

Example:

$E \rightarrow iE'$

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

code:

```
E() {
    if( look_ahead == 'i')
    {
        match('i');
        E'();
    }
}
E'() {
    if (look_ahead == '+') {
        match('+');
        match('i');
        E'();
    }
    else
        return;
}
match(char c) {
    if( look_ahead == c ) {
        look_ahead = getnextchar();
    else
        print("ERROR");
}
main() {
    E();
    if( look_ahead == '$' ) print("parsing successful!");
}
```

Phase 3:

Semantic analysis

- Lexical analyzer gives stream of tokens to Syntax analyzer / parser which generates parse tree
- Semantic analyzer adds additional information in the parse tree for type checking.

→ Syntax directed definition (SDD):

- Attributes are associated with grammer symbol & semantic rules are associated with productions

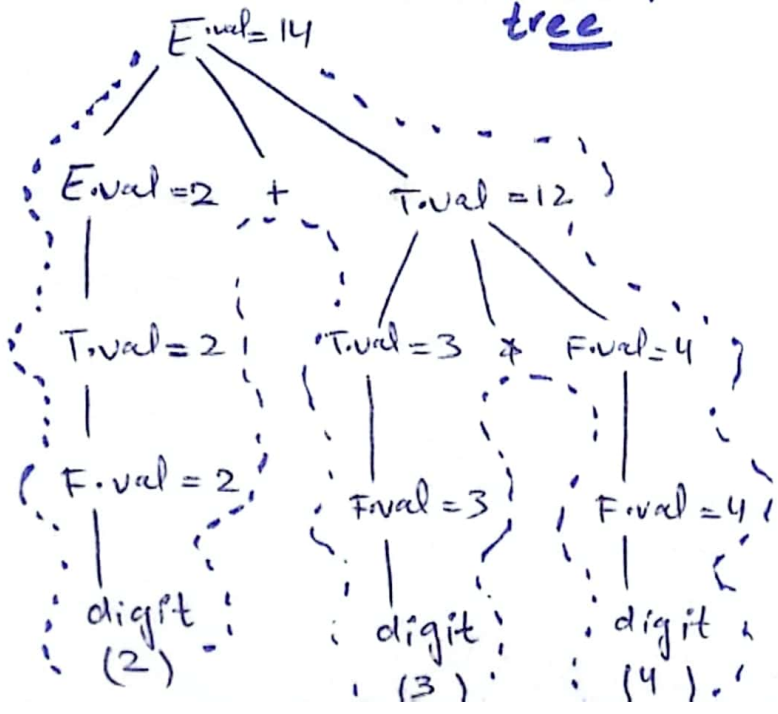
Attributes may be number, string, reference, datatype etc

Grammer + Semantic rules

production	Semantic Rule
$E \rightarrow E+T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow$ digit	F.val = digit.lval (lexical value)

2 + 3 x 4

Annotated parse tree



- we can observe associativity and precedance by just looking at Grammer
- left recursive = left associative
- symbol farthest from start symbol has highest precedance

Example:

production	Semantic Rule
$E \rightarrow E \# T$	$E.val = E.val \times T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T \& F$	$T.val = T.val \_\_ F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow digit$	$F.val = digit.lval$

If the expression "8#12&4#16&12#4&2" is evaluated to 512 then which is best replacement to achieve this result

a) T.val = T.val x F.val

b) T.val = T.val + F.val

c) T.val = T.val − F.val

d) None

Solution:

Instead of replacing each in parse tree, we can observe that Grammer is left recursive for '#' and '&', so, it is left associative. & has highest precedance than # due to farthest from start symbol.

→ Trying option (a)

$8 \# (12 \& 4) \# (16 \& 12) \# (4 \& 2)$

$\# = \times$

$\& = \times$

$8 \times (12 \times 4) \times (16 \times 12) \times (4 \times 2) \neq 512$

→ Trying option (b)

$\# = *, \& = +$

$8 \times (12+4) \times (16+12) * (4+2) \neq 512$

→ Trying option (c)

$\# = *, \& = -$

$8 \times (12-4) \times (16-12) \times (4-2) = 512$

So (c) is correct!

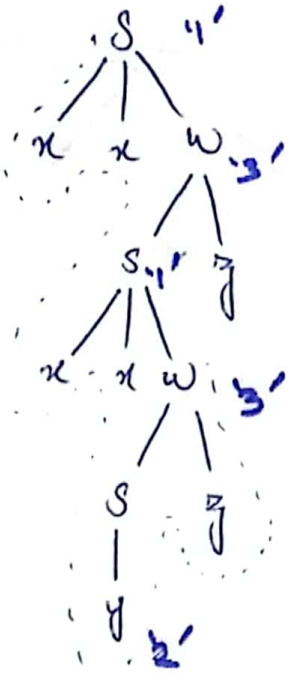
Example:

	Semantic Rules
$S \rightarrow xxw \mid y$	{print('1')} | {print('2')}
$w \rightarrow Sz$	{print('3')}

Perform semantic analysis on "xxxxyzz".

Sol:

Parse Tree:



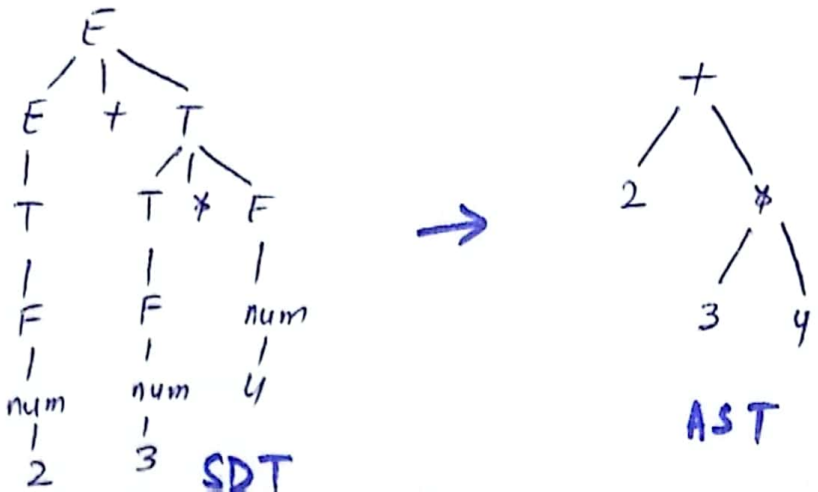
when rule is finished during traversal, apply corresponding semantic rule.
For example, **S→y** is completed first during traversal and semantic rule is **print('2')**, so '2' is produced and traversal is continued till reached to starting node

= 23131

Abstract Syntax Tree (AST):

- After removing irrelevant information from parse tree, AST is generated.

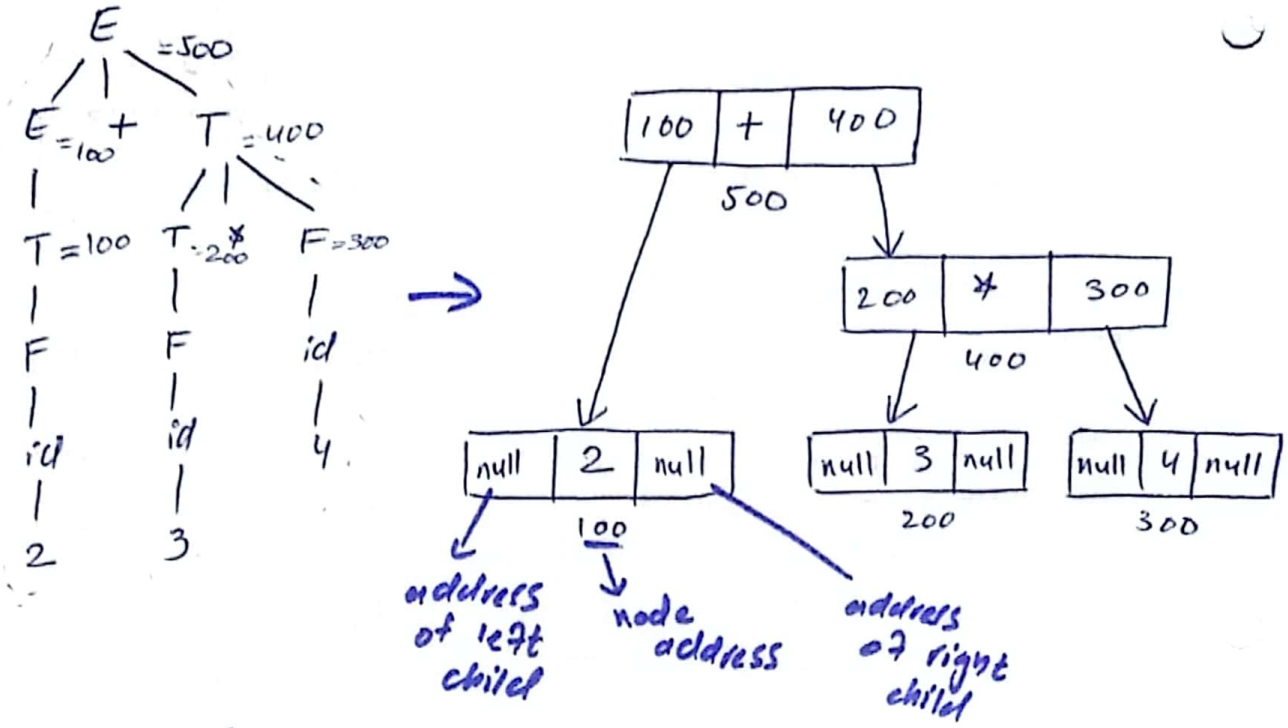
For example:



Example:

	Semantic Rule
$E \rightarrow E+T$	{ E.nptr = newNode(E.nptr, '+', T.nptr) }
| T	{ E.nptr = T.nptr }
$T \rightarrow T * F$	{T.nptr = newNode(T.nptr, '*', F.nptr)}
| F	{T.nptr = F.nptr}
$F \rightarrow id$	{ F.nptr = newNode(null, id_val, null) }

perform semantic analysis & generate result on "2+3*4"



This semantic rules generate Abstract syntax tree (AST) or syntax directed tree (SDT).

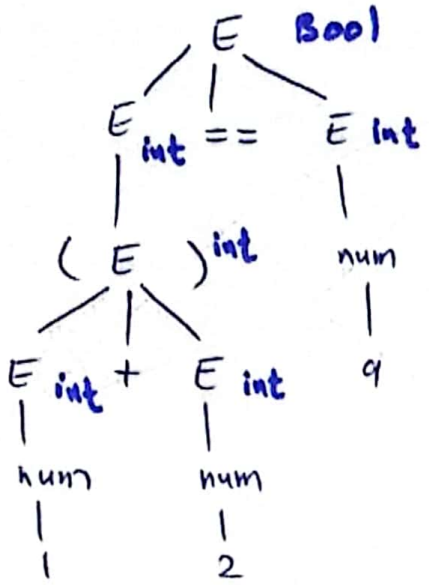
Example:

	Semantic Rules
$E \rightarrow E+E$	{if(E.type == E.type) && (E.type=int) then E.type=int else error}
| E == E	{if(E.type == E.type) && (E.type = int/Bool) then E.type = Bool else error}
| (E)	{ E.type = E.type }
| num	{ E.type = int }
| True	{E.type = Bool}
| False	{ E.type = Bool }

what will be type of (5+7) == (1+2) ?

(1+2) == 9

what will be type returned by semantic rules?



'Bool' will be final type.

Types of SDD

① S-Attributed

- A SDD that uses only synthesized attribute is called S-Attributed SDD.

Example:

$A \rightarrow BCD$

$A_i = B_i$

$A_i = C_i$

$A_i = D_i$

Synthesized attributed:- child node value is consumed/used by parent node

- Semantic actions are placed on right most side of CFG
- Attributes are evaluated bottom up because child value is accessed by parent node.

② L-Attributed

- A SDD that uses both inherited and synthesized attribute is L-attributed. But each inherited attribuded is restricted to inherit from parent or left sibling because DFS traversal is used.

'C' can only inherit attribute from A & B not D.

Inherited attributed:- child inherit attributes from parent node

- Semantic rules are placed anywhere on RHS of productio
- Depth first search from left to right is used.

Example:

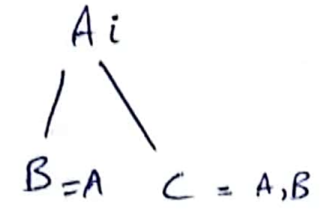
Identify productions are S-attributed, L-attributed or both

1) $A \to BC$ $\{ Bi = Ai , Ci = Bi , Ai = Ci \}$

2) $A \to QR$ $\{ Ri = Ai , Qi = Ri , Ai = Qi \}$

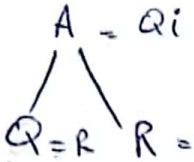
3) $A \to PQ$ $\{ Ai = Pi , Ai = Qi \}$

(1)



As 'C' can inherit from A & B, parent & left sibling, it is L-attributed

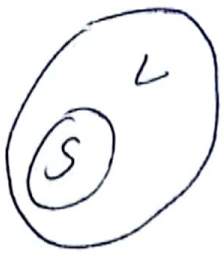
(2)



none of the both!

(3)

both S & L



if attribute is 'S', it will also be 'L' attributed

Example:

Given Grammer converts binary to decimal:

$S \to L$ $\{ S.dv = L.dv \}$

$L \to LB$ $\{ L.dv = 2 \times L.dv + B.dv \}$

$L \to B$ $\{ L.dv = B.dv \}$

$B \to 0$ $\{ B.dv = 0 \}$

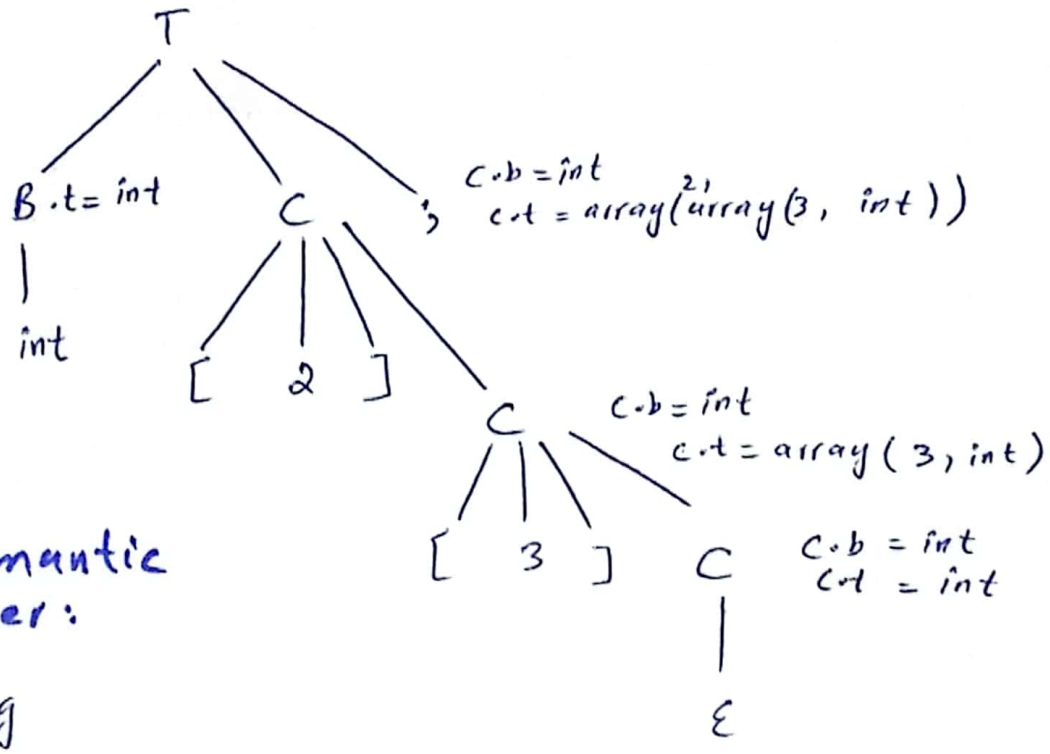
$B \to 1$ $\{ B.dv = 1 \}$

create annotated parse tree for "1010"

Write CFG & Semantic rule for given C statement

```
int [2][3];
```

	Semantic Rules
$T \rightarrow BC;$	$\{T.t = C.t,\ C.b = B.t\}$
$B \rightarrow int$	$\{B.t = int\}$
$B \rightarrow float$	$\{B.t = float\}$
$C \rightarrow [num]C$	$\{C.t = array(num.val, C_1.t)\}$ $\{C.b = C.b\}$
$C \rightarrow \varepsilon$	$\{C.t = C.b\}$



Tasks of Semantic analyzer:

1) Type checking
2) Store variable type in symbol table
3) Create syntax tree
4) Evaluate arithematic expression
5) Infix to postfix or prefix
6) Conversion from binary to decimal
7) Counting # of reductions

Example:

Given grammer converts floating binary to decimal

$S \rightarrow L.L \quad \{S.dv = L.dv + L.dv / 2^{L_2.nb}\}$

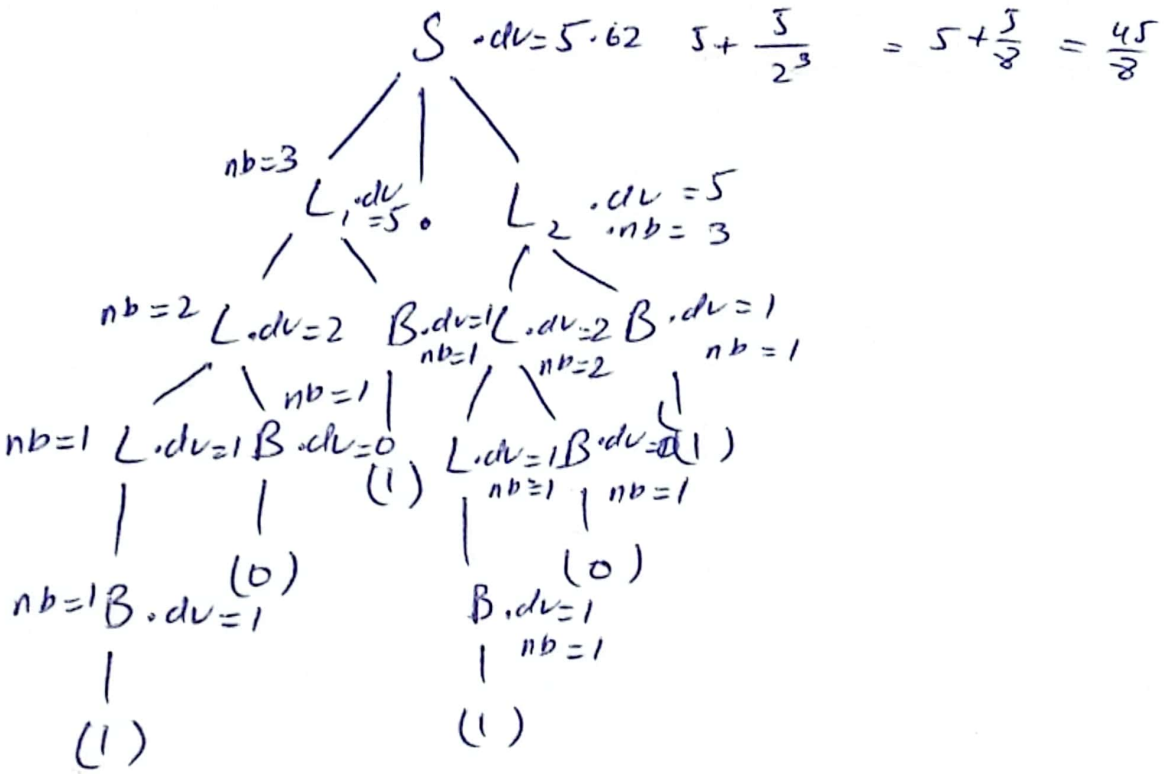
$L \rightarrow L.B \quad \{L.dv = 2 \times L.dv + B.dv,\ L.nb = L.nb + B.nb\}$

$L \rightarrow B \quad \{L.dv = B.dv,\ L.nb = B.nb\}$

$B \rightarrow 0 \quad \{B.dv = 0,\ B.nb = 1\}$

$B \rightarrow 1 \quad \{B.dv = 1,\ B.nb = 1\}$

create annotated parse tree for 101.101



SDT to Generate 3 address code (3AC):

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

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

$E \rightarrow T$ { E.place = T.place }

$T \rightarrow T * F$ { T.place = new temp(); gen(T.place = E.place + T.place }

$T \rightarrow F$ { T.place = F.place }

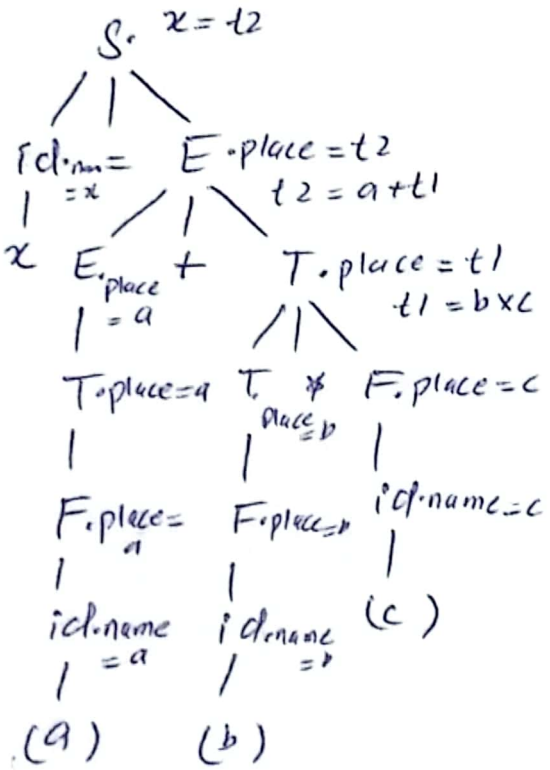
$F \rightarrow id$ { F.place = id.name }

In 3AC, maximum 3 addresses are allowed like in this expression $a = b + c \times d$, addresses are 4 and need to create a temporary variable to convert it to 3AC.

3AC is used in intermediate code generation.

Lets parse and generate 3AC of

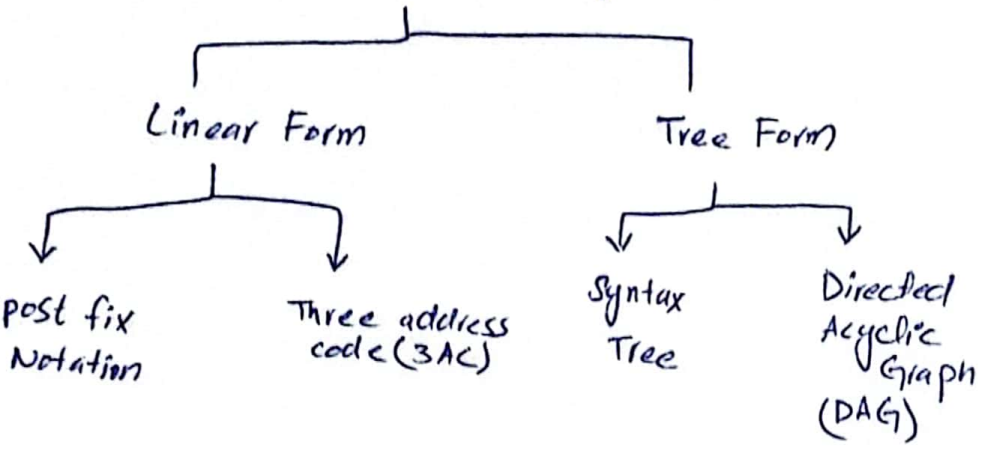
x = a + b x c ;

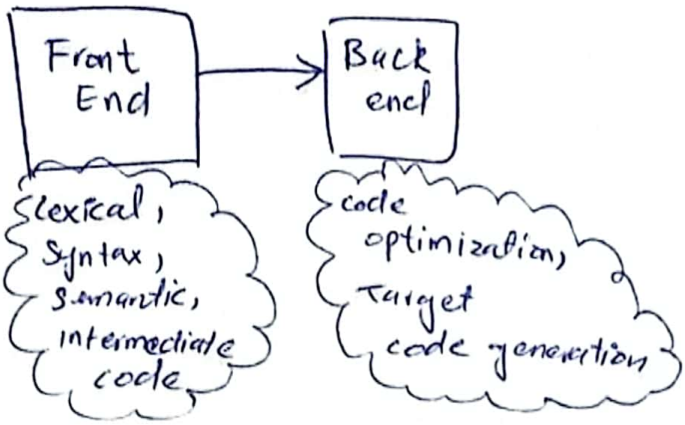


3AC

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

Phase 4:
"Intermecliate code generation"





➔ If i want to design compiler for windows & Linux os, then I just need to manipulated Backend part while front end will be same for all machines. It means we do not have to write compiler from scratch. Compiler construction tools like Lex, Yacc, bison are also used to design front end. we just need to decide the structure & rules of our language.

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

postfix = a b + a b + c + *

3AC =
t1 = a+b
t2 = a+b
t3 = t2+c
t4 = t1 * t3

Syntax Tree:

DAG: Remove duplicate parent-child subtree

→ Types of 3AC statements

Try to put 3 address per instruction

1) $x = y \ op \ z$

op = binary operator {+, −, *, /, ...}

Example: $x = a + b * c$

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

2) $x = op \ z$

op = unary operator {++, --}

3) $x = y$

4) if (x relop y) goto L (conditional jump)

relop = relational operator {<=, ==, >=, !=, ...}

5) goto L (unconditional jump)

6) A[i] = x

7) $x = *p$

$y = \&x$

→ Representation of 3AC:

1) Quadruple 2) Triples 3) Indirect triples

Example $-(a * b) + (c * d) + e$

3AC:

```
t1 = a * b
t2 = -t1
t3 = c * d
t4 = t3 + e
t5 = t2 + t4
```

1) Quadruple:

[Consider SAC of previous example]

	operator	op1	op2	result
0	*	a	b	t1
1	-	t1		t2
2	*	c	d	t3
3	+	t3	e	t4
4	+	t2	t4	t5

- Require extra space
- Re-ordering of instructions is possible if there is no inconsistency in result occurs.

2) Triples:

- Remove last column from Quadruple
- put index of table in place of temporary variables {t1, t2, .. t5}

	operator	op1	op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

- Requires less space than Quadruple
- Reordering is not possible due to table index as address

(1) → t2 entry no. 1 in table of Quadruple

(3) → t4, entry no. 3 in Quadruple last column latest t4

3) Indirect Triples

100	(0) → index no. 0 of table of "Triples"
101	(1)
102	(2)
103	(3)
104	(4)

- Requires extra memory address
- Re-order of instruction is possible

Back patching

Leaving table as empty and filling them later is backpatching.

we have seen 7 types of instruction in programming

For if-else, 3AC will be:

Example:

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

3AC:

Line#

1. if (a<b) goto 4
2. t=0
3. goto 5
4. t = 1

Example 2:

if (a<b) && (c<d) then t=1 else t=0

1 if (a<b) goto 4

2 t=0

3 ~~if c~~ goto 7

4 if (c<d) goto 6

5 goto 2

6 t=1

7

Loop : (3AC)

```
for( i=1 ; i<n; i++ ){
    x = a+b*c ;
}
```

3AC

```
1   i = 1
2   if(i<n)  goto 4
3     goto 9          // false case
4     t1 = b*c
5     t2 = a+t1
6     x = t2
7     i = i+1
8     goto 2
```

OR

```
1   i = 1
2   if(i>n)  goto 8
3   t1 = b*c
4   t2 = a+t1
5   x = t2
6   i = i+1
7   goto 2
8
```

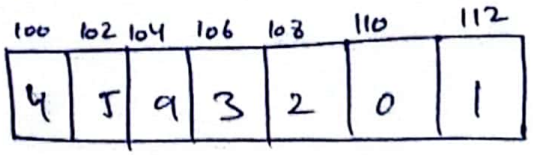
Switch Case:

```
switch(i) {
  case 1:
     x1 = a1+b1*c1;
     break;
  case 2:
     x2 = a2+b2*c2; break
  default:
     x3 = a3+b3*c3;
     break;
}
```

3AC

```
1   if(i==1)  goto 7
2   if(i==2)  goto 11
3   t1 = b3*c3;
4   t2 = a3+t1;
5   x3 = t2
6   goto 14
7   t1 = b1*c1
8   t2 = a1+t1
9   x = t2
10  goto 14
11  t1 = b2*c2
12  t2 = a2+t2
13  x = t2
14
```

Arrays: (1D)



Size of int = 2

$A[3] = \text{base\_address} + (i - \text{lower bound}) \times \text{size of single element}$

$= 100 + (3-0) \times 2$

$= 106$

Example:

```
int A[10], B[10];
int x=0; i;

for(i=0; i<10; i++){
        x = x + A[i] * B[i];
}
```

3AC:

```
1   x=0
2    i=0
3    if(i ≥ 10)  goto 15
4     t1 = base address of A
5     t2 = i*4
6     t3 = t1[t2]
7     t4 = base address of B
8     t5 = i*4
9     t6 = t4[t5]
10    t7 = t6*t5
11    t8 = x+t7
12    x = t8
13    i = i+1
14    goto 3
15
```

2D Array :

$$A[3][3] = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 0 & 1 \\ 1 & 2 & 5 \end{bmatrix}$$

$A[i][j]$ = Base_address + (i x # of columns + j) x size of datatype

$A[1][3] = 0 + (1 \times 3 + 3) \times 4$

$= 24$

```
int i=10,j=15;
X = A[i][j]
```

```
t1 = i*15
t2 = t1+j
t3 = t2*2
t4 = Base address of A
t5 = t4[t3]
X = t5
```

or

```
t1 = base addr of A
t2 = ix4
t3 = t1[t2]
t4 = jx4
t5 = t3[t4]
X = t5
```

3D Array :

```
int A[a][b][c];
```

$A[i][j][k] = A[i \times b \times c + j \times c + k] \times$ size of datatype

Example

Give 3AC code, identify correct option. assume array of char is 8 bits & integer 32 bit ≈ 4 bytes

```
t0 = ix1024
t1 = jx32
t2 = k*4
t3 = t1+t0
t4 = t3+t2
*t5 = X[t4]
```

a) int X[32][32][8];

b) int X[4][1024][32];

c) char X[4][32][8];

d) char X[32][16][2];

Try option (a)

$$X[i][j][k] = X[i \times 32 \times 8 + j \times 8 + k] \times 4$$

~~$X[i \times 1024 + j \times 32 + k]$~~ -

$$= X[i \times 32 \times 8 \times 4 + j \times 8 \times 4 + k \times 4]$$

$$= X[i \times 1024 + j \times 32 + k \times 4]$$

So option (a) is right one!

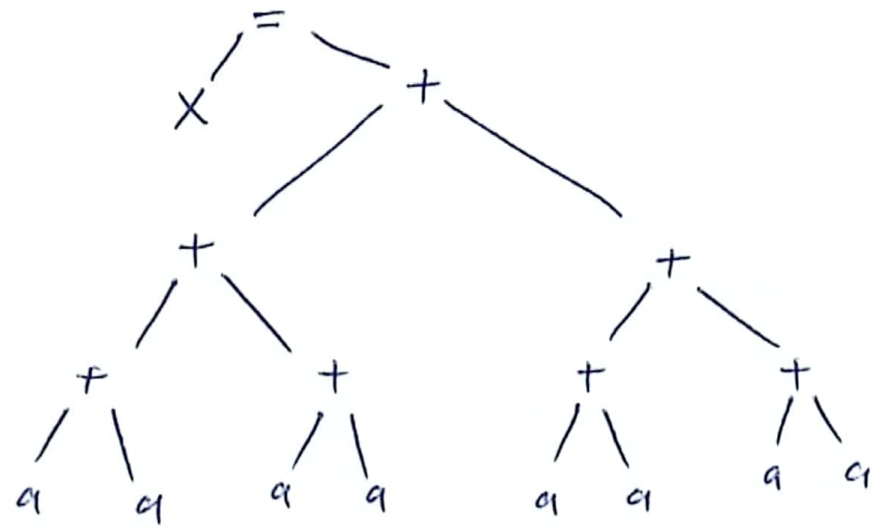
Syntax tree & DAG :

(directed acyclic graph)

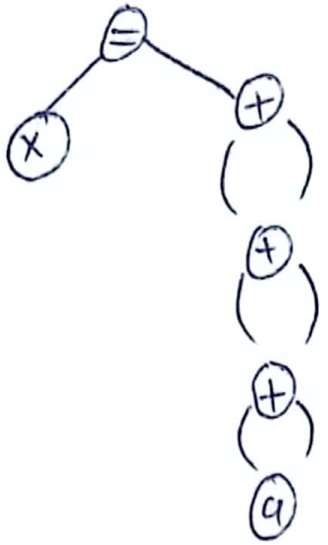
Example:

$$X = ((a+a)+(a+a)) + ((a+a)+(a+a))$$

Syntax tree

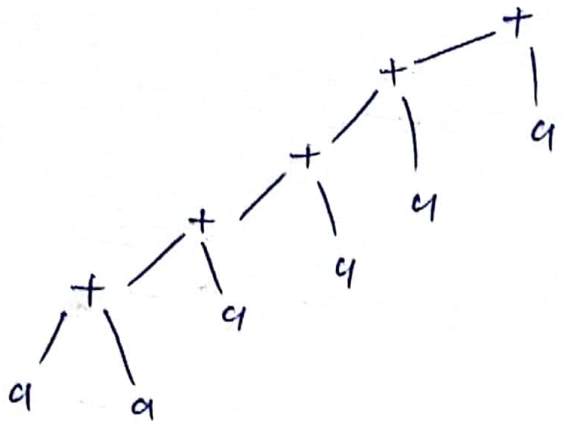


DAG : Remove same subtree & create a loop



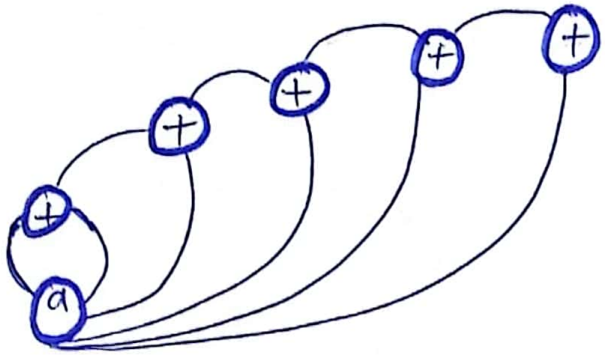
2) X = a + a + a + a + a + a

Syntax tree:



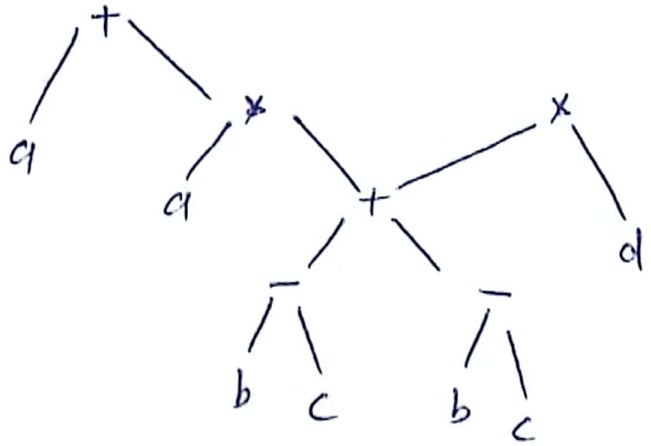
DAG:

=

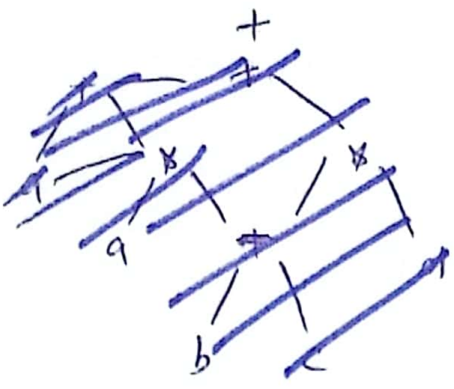


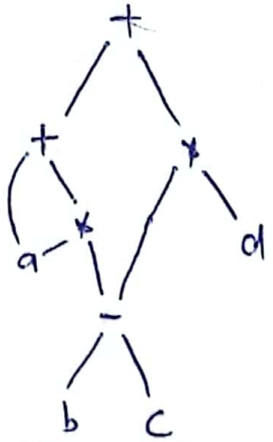
3) X = a + a * (b - c) + (b - c) * d

Syntax Tree



DAG:





4)

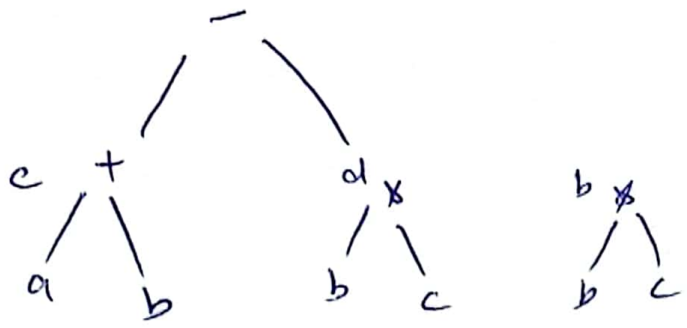
$d = b * c$

$e = a + b$

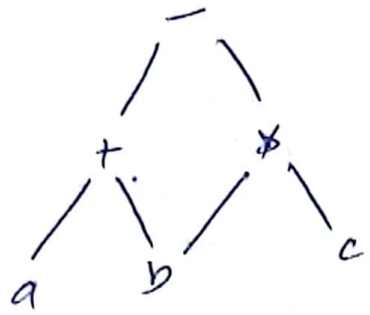
$b = b * c$

$a = e - d$

Syntax Tree:



DAG:



5)

~~$d = b * c$~~

~~$e = a + b$~~

~~$b = b * c$~~

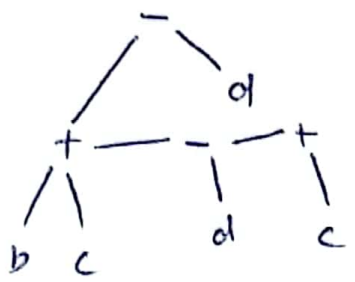
$a = b + c$

$b = a - d$

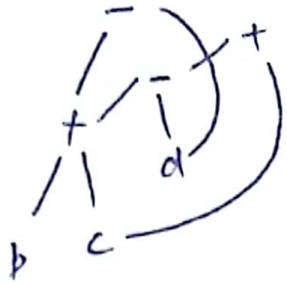
$c = b + c$

$d = a - d$

Syntax Tree:



DAG



Example : 6

$a = b+c$

$c = a+d$

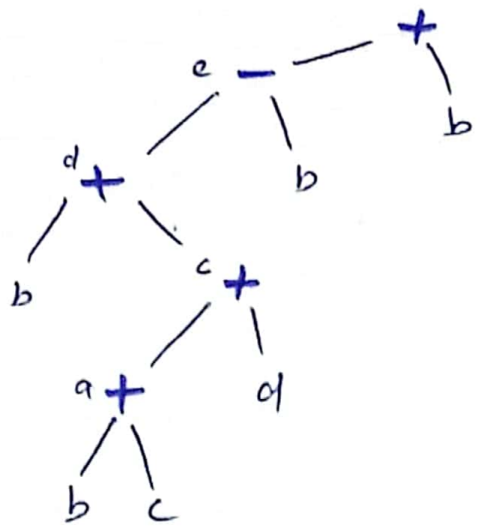
$d = b+c$

$e = d-b$

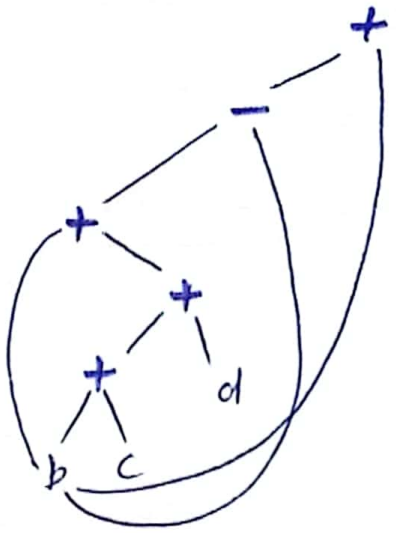
$a = e+b$

find min # of nodes in DAG.

Syntax tree



DAG:



Reducing by putting values in equation (substitution):

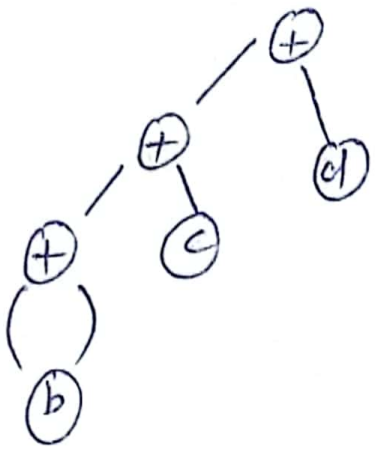
$a = e+b$

$= d-b+b$

$= b+c$

$= b+a+d$

$= b+b+c+d$



min # of nodes = 6

" edges = 6

3AC for procedures:

```
n = f(a[i]);

t1 = Base address of a
t2 = i x 4
t3 = t1[t2]
param t3
t3 = call f, 1
n = t3
```

SDT to generate 3AC for Control Flow statements

$P \rightarrow S$	S.next = newlabel(); P.code = S.code
$S \rightarrow$ assign	S.code = assign.code
$S \rightarrow$ if (B) S_1	B.true = newlabel(); B.false = S_1.next = S.next S.code = B.code || label(B.true) || S_1.code
$S \rightarrow$ if (B) S_1 else S_2	B.true = newlabel(); B.false = newlabel(); S_1.next = S_2.next = S.next S.code = B.code || S_1.code || S_2.code || gen(goto S.next) || label(B.false)
$S \rightarrow$ while(B) S_1	begin = newlabel() B.true = newlabel() B.false = S.next S_1.next = begin S.code = label(begin) || B.code || S_1.code

|| label(B.true) || gen(goto begin)

$S \rightarrow S_1 S_2$ $S_1.next = newlabel()$

$S_2.next = S.next$

$S.code = S_1.code \,||\, S_2.code \,||\, label(S.next)$

$B \rightarrow B_1 \,||\, B_2$ $B_1.true = B.true$

$B_1.false = newlabel();$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \,||\, B_2.code \,||$ label($B_1.false$)

$B \rightarrow B_1 \,\&\&\, B_2$ $B_1.true = newlabel()$

$B_1.false = B.false$

$B_2.true = B.true$

$B_2.false = B.false$

$B.code = B_1.code \,||\, B_2.code \,||$ label($B_1.true$)

$B \rightarrow !\, B_1$ $B_1.true = B.false$

$B_1.false = B.true$

$B.code = B_1.code$

$B \rightarrow E_1$ relop E_2 $B.code = E_1.code \,||\, E_2.code$ || gen('if' $E_1.addr$ relop.op $E_2.addr$ 'goto' B.true) || gen('goto' B.false)

$B \rightarrow true$ $B.code = gen('goto'\ B.true)$

$B \rightarrow false$ $B.code = gen('goto'\ B.false)$

Example:

Write SDT to compute data type and size of array. Consider int occupies 4 bytes & float 8 bytes

$T \rightarrow B\ C$ { t = B.type ; w = B.width;
T.type = C.type ; T.width = C.width; }

$T \rightarrow$ int { T.type = integer ; T.width = 4
| float T.type = float ; T.width = 8 }

$C \rightarrow \varepsilon$ { C.type = t ; C.width = w }

$C \rightarrow [num]\ C_1$ { C.type = array (num.value , C.type);
C.width = num.value x C_1.width }

Phase 5: Code Optimization

→ Machine Independent

1) Loop optimization
   a) Code motion or frequency reduction
   b) Loop unrolling
   c) Loop jamming
2) Folding
3) Redundancy elimination
4) Strength reduction
5) Algebraic Simplification

→ Machine dependent

1) Register allocation
2) Instruction selection
3) Use of Addressing modes
4) Peephole optimization
   a) Redundant LOAD/STORE
   b) Flow of Control optimization
   c) Use of machine idioms

Loop Optimization:

1) To apply loop optimization, we must first detect loops.
2) For detecting loops, control flow analysis via control Flow Graph (CFG) is used.
3) To build program flow graph, we need to find basic blocks
4) A basic block is a sequence of 3AC statements where control enters at the begining & leaves only at the end without any jump or halt.

Construction of program flow graph:

1) Basic block:

Intermediate code generator
↓
3 address code
↓
Loop identification via
control flow analysis
using program flow graph

→ In order to find basic blocks, we need to find leaders in 3AC

→ A basic block will start from one leader to next leader but not including next leader

* Finding leaders in a basic block:

① 1st 3AC instruction in intermediate code is leader

② Any instruction that is the target of conditional goto or unconditional goto is a leader

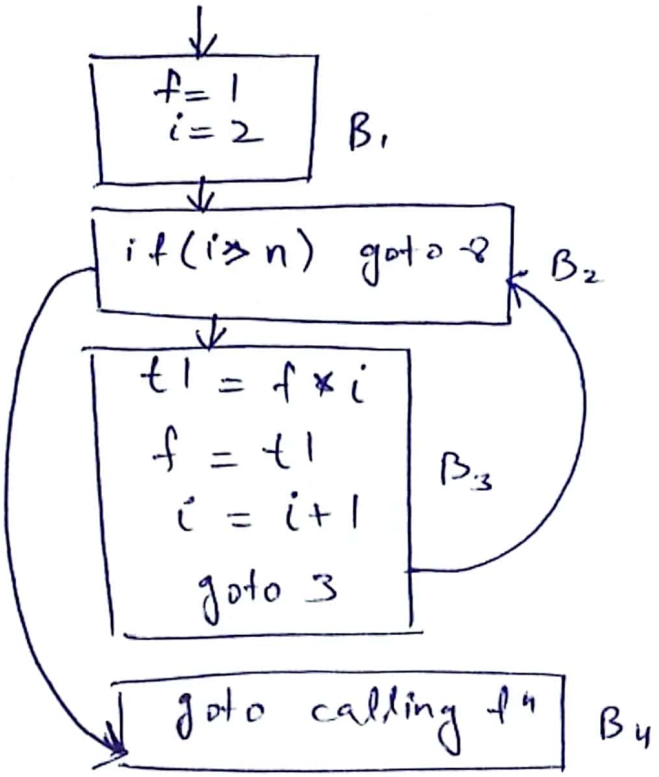
```
if(x<3) goto 3
// '3' is leader
```

③ Any instruction that immediately follows a conditional or unconditional jump is a leader.

```
x = y+2.    // leader
if(x==0) goto _
```

Example:

```
*  1   f = 1
   2   i = 2
*  3   if (i > n) goto 8
*  4   t1 = f * i
   5   f = t1
   6   i = i + 1
   7   goto 3
*  8   goto calling fn
```



→ If loop exist, there will be cycle in the graph.

→ Optimization can be performed at each block basic

Example 2:

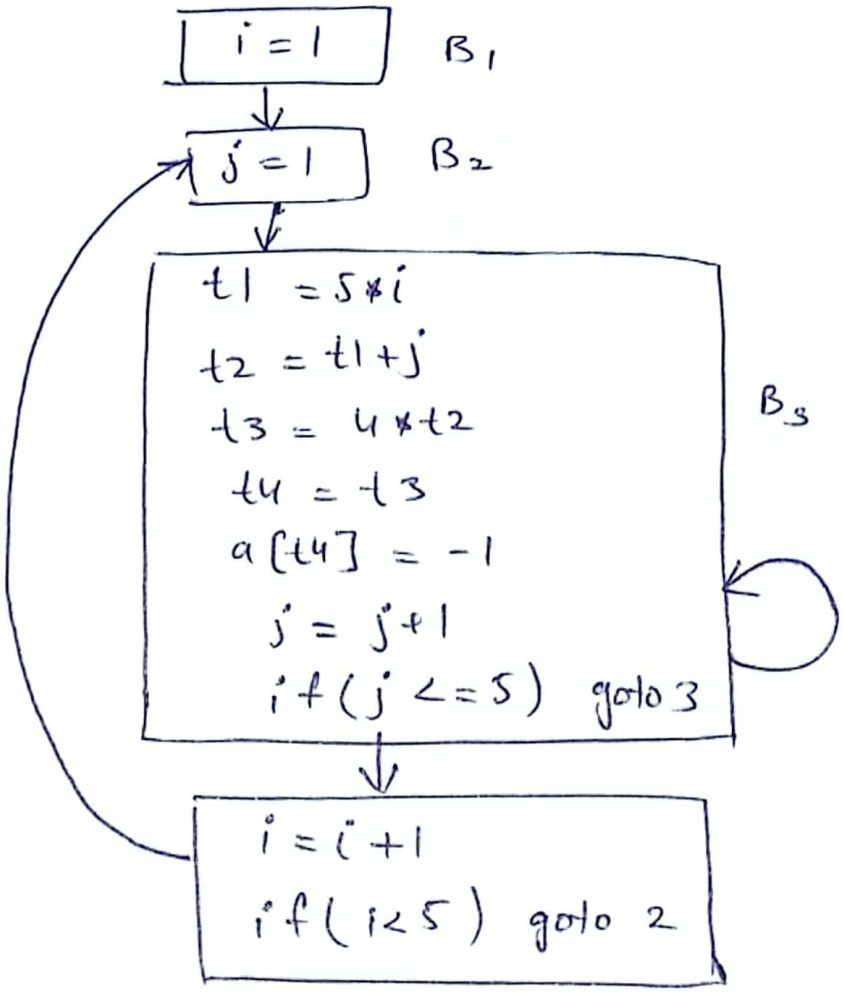
```
*  1    i = 1
*  2    j = 1
*  3    t1 = 5 * i
   4    t2 = t1 + j
   5    t3 = 4 * t2
   6    t4 = t3
   7    a[t4] = -1
   8    j = j + 1
#  9    if (j <= 5) goto 3
*  10   i = i + 1
#  10   if (i < 5) goto 2
```

Construct program flow graph.

Identify leader

// basic block

// ~~Cont~~ Construct Graph



Loop optimization

① Frequency Reduction: (code Motion)

A statement or expression which can be moved outside the loop without affecting the semantics of program

```
int p;
char *name = "Aoun";
for(int i=0; i< len(name); i++){
   print(name[i]);
    p=0;
}
```

Problems

'len' will be invoked 5 times, p=0 is constant expression & can be kicked out.

② Loop unrolling:

Reducing # of times comparisons are made in the loop.

```
for(i=0; i<10 ; i++){
   print ("Mr. Aoun");
}
```

→

```
for(i=0; i<10; i+=2){
   print("Mr. Aoun");
   print("Mr. Aoun");
}
```

③ Loop jamming:

Combining bodies of multiple loops

```
for(i=0; i<5 ; i++){
   a = i+5;
}
for(i=0; i<5 ; i++){
   b = i+10;
}
```

→

```
for(i=0; i<5; i++){
   a = i+5;
   b = i+10;
}
```

Dead code:

→Unreachable code
→code never executes or used or participate in final outcome
For example, In if-else condition, 'if' is always executed then 'else' block is dead block.

① Unused variable, functions
② Redundant code
③ constant folding

```
int x = 10*5 + 3;
```

here final outcome can be stored in 'x'

Induction variables & strength reduction

- Variable increasing or decreasing by a constant factor
  t4 = i x 4

is induction variable

- strength reduction is to replace ~~ster~~ strong operator like '*' with cheap operator '+'

Liveness analysis:

Register allocation:
variables are stored in registers. It makes problem when # of program variables are greater than available registers. we need to synchronize some variables in some way. purpose of register allocation is to assign a single register to multiple variables without changing program behaviour/semantics

X is live at statement S_i if

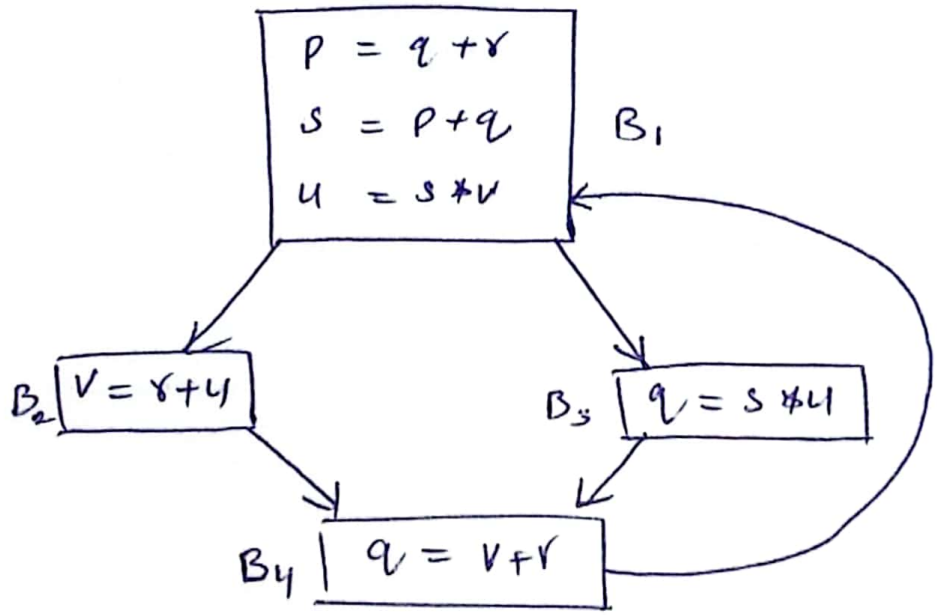
1) There is a statement S_j using X
2) There is a path from S_i to S_j
3) There is no new definition to X before S_i

1. $X = a + b$
2. $Y = d + c$
3. $X = X + b$
4. $a = b + d$
5. $X = a + b + c$

	a	b	c	d	X	Y
1	L	L	L	L	D	D
2	D	L	L	L	L	D
3	D	L	L	L	L	D
4	D	L	L	L	D	D
5	L	L	L	D	D	D

Example 2:

perform liveness analysis:



→ check 3 conditions of liveness

	P	q	r	s	u	v
2	D	D	L	D	L	D
3	D	D	L	L	L	L

Example 3:

Assume that all operations take their operands from register, what is the minimum # of registers needed to execute this code without spilling?

↓

Storing variables in main memory when # of register are allocated to variables too much!

```
a = 1
b = 10
c = 20
d = a + b
e = c + d
f = c + e
b = c + e
e = b + f
d = 5 + e
```

a) 2 b) 3 c) 4 d) 6

Hint: perform liveness analysis.

Sol:

~~a = 1~~ r1
~~b = 10~~ r2
~~c = 20~~ r3
~~d = a + b~~ r1 // 'a' is dead, reclaim 'r1'
~~e = c + d~~ r1 // 'd' is dead, reclaim 'r1'
~~f = c + e~~ r2 // 'b' is dead
~~b = c + e~~ r1 (c: r3, e: r2)
~~e = b + f~~ r1 (b: r1, f: r2)
~~d = 5 + e~~ r1 (e: r1)

Registers used = r1, r2, r3

So, minimum 3 are required

Peephole optimization
(machine dependent)

performed during code generation

a) Redundant LOAD/STORE elimination:

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

LOAD   R0, b
LOAD   R0, c
STORE  a, R0
LOAD   R0, b
ADD    R0, c
STORE  a, R0   ]
LOAD   R0, a   ]  useless instructions
ADD    R0, e
STORE  d, R0
```

b) Flow of control optimization:

① Avoid jumps on jumps

```
L1:
    jump L2
    ==
L2:
    jump L3
    ==
L3:
    jump L4
    ==
L4:
    x = a+b/c
```

→

```
L1:
    jump L4
    ==
L4:
    x = a+b/c
```

② Eliminate dead code

```
int i = 0;
if(i == 1)
print("COMSATS")
```

c) use of M/C idioms:

i = i + 1;

```
LOAD   R0, i
ADD    R0, #1      ->   INC i
STORE  i, R0
```