- HIGH LEVEL LANGUAGE → COMPILERS (program) → LOW LEVEL LANGUAGE / MACHINE CODE

[ASSEMBLY LANGUAGE]

- BATCHWISE PROCESSING OF PROGRAMS

↓

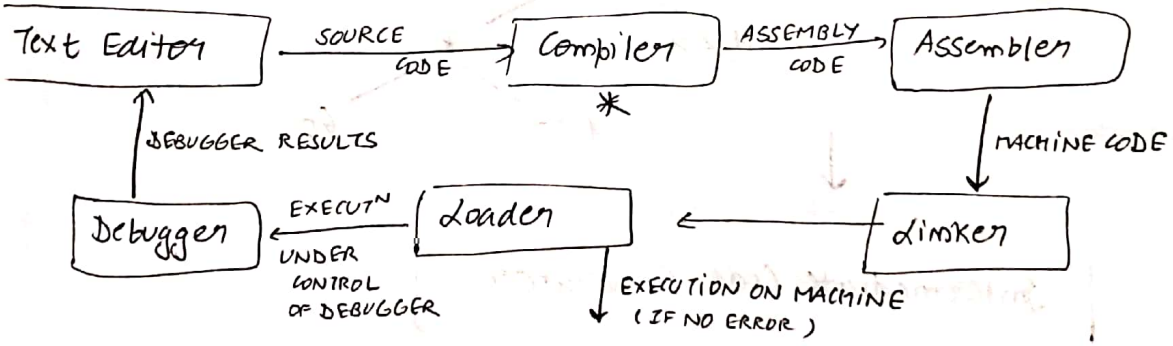
[PUNCH CARDS]

- JOHN BACKUS → COMMERCIAL COMPILERS → 1957 → 18 PERSON YRS.

FORTRAN - I

- Assemblers are usually built into compilers
- compilers should have foll. criteria
  1) convert one representation to another
  2) Meaning should remain the same

THE BIG PICTURE



* Compiler is a Translator

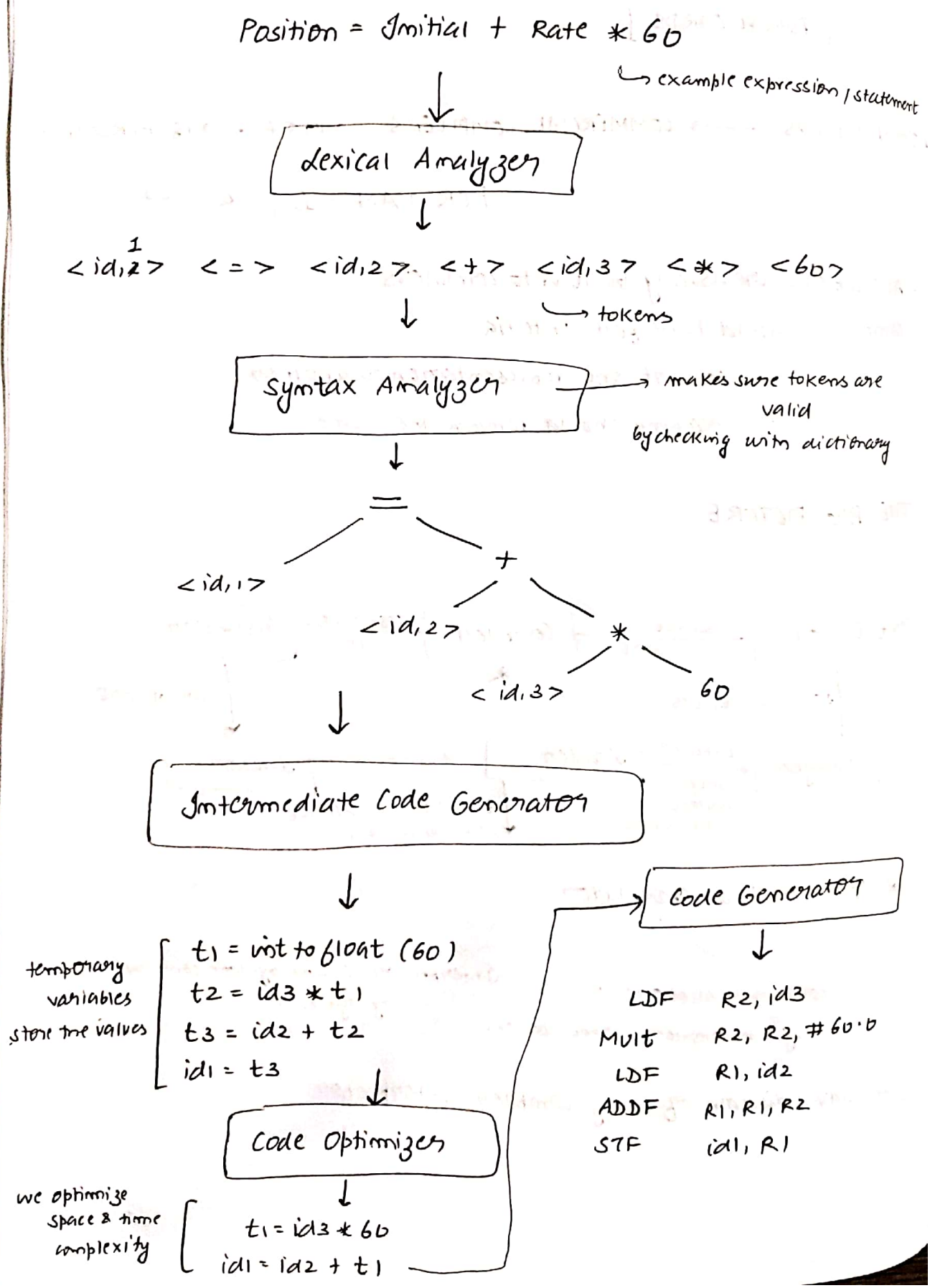
Compiler (all at once conversion)
eg: C# compiler, TurboC, Dev C++

Interpreter (Line-by-line conversion) ...
eg: Python, R

-HW adv., disadv. of using compiler, interpreter

- we must know about variables, expressions, constants. In presence of compiler, registers, memory etc. are not to be remembered while programming

PHASES OF A COMPILER



- yacc, lex are used to build a compiler

COMPILERS : PRINCIPLES, TECHNIQUES & TOOL , ALFRED V ALI O, MONIEN, ULLMAN

- each word in expression is mapped to symbol table by lexical analyzer during token generation.

ADVANTAGES OF COMPILER OVER INTERPRETOR

- Traditional compiler translates source code into directly executable machine instructions., once & for all.
- since compilation is one timer, compilers usually put more effort into optimizations & thus execute faster even after translation.
- compilers can perform global optimizations.
- Code uses less CPU cycles, ∴ runs faster & uses less power

DISADVANTAGES OF INTERPRETER

- An Interpreter reads character by character & has to re-do all the work every time user wants to run a program through it
- Interpreters translate only the part of program it is currently executing, thus they can't perform global optimizations.
- Many bugs (like type error) are encountered by an interpreted program at run-time, ~~thus more bugs are delivered to customer~~

ADVANTAGES OF INTERPRETER OVER COMPILER

- source code is multi-platform, compiled code is processor specific
- It's usually easier to develop & debug in an interpreted environment
- Interpreters are easier to use, particularly for beginners, since errors are immediately displayed.

DISADVANTAGES OF COMPILER

- Hardware specific source code, ∴ multiple versions of source code must be maintained.
- large applicat^n can take significant amt of time to compile.
- difficult to debug code that is optimized.

LEXICAL ANALYSIS #1

Generates a series of tokens

→ identifiers → variables, keywords
operators
whitespaces
Tabs

int a;b;c;

SYNTAX ANALYSIS #2

Analysis of structure

we is going to school → syntactically correct, but not semantically

int a, b, c → analysis is done using parse tree

SEMANTIC ANALYSIS #3

understanding the meaning

int a = b + c ;
↑ ↑ ↑
int float char
↑ ↑
typechecking

] Removes all the code ambiguity

eg Bytecode generation in Java

INTERMEDIATE CODE GENERATION (3 ADDRESS CODES) #4

OPTIMIZATION (OPTIONAL) #5

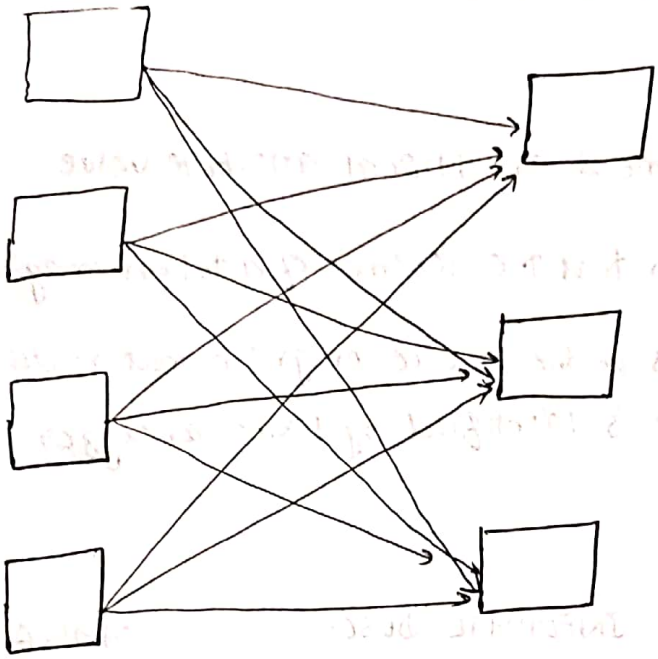
TARGET MACHINE CODE GENERATION #6

(Assembly lang. code)

#1, #2, #3 come under analysis phase (machine independent) / front end

#5, #6 comes under synthesis phase / back end (m/c dependent)

4 SOURCE LANG. 3 TARGET MACHINES



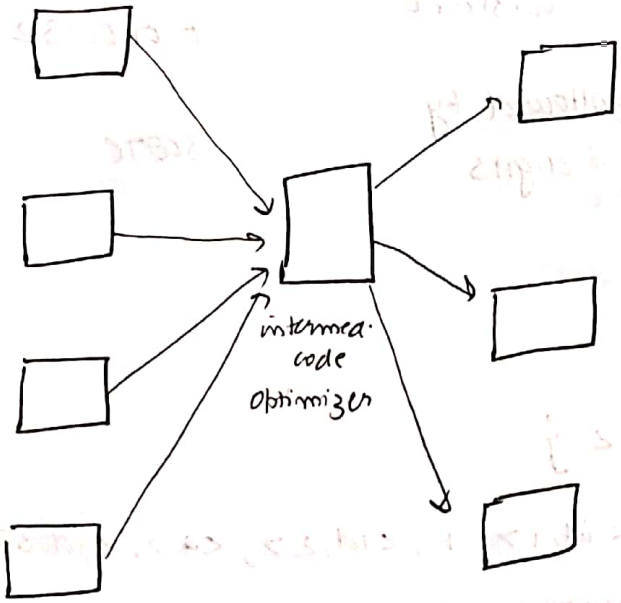
If there was no intermediate code generation, then we need

4 frontends +

4x3 optimizers +

12 code generators

4 SOURCE LANG. 3 TARGET M/C



In presence of interme-code generation, we need

4 frontends +

1 code optimizer +

3 code generators

LEXICAL ANALYSIS

TOKEN: It is a pair of token name & an optional attribute value

PATTERN: A description of the form that the lexemes of a token may take.

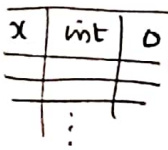
LEXEMES: A sequence of characters in the source program that matches the pattern for a token & is identified by lexical analyzer as an instance of a token.

EXAMPLE •	TOKEN	INFORMAL DESC.	SAMPLE LEXEME
1)	if	characters i, f	if
2)	number	any numeric constant	3.14159 0.0000132
3)	identifier	letter followed by letters & digits or	score

- $x = a + b * 2$

LEXEMES : { x, =, a, +, b, *, 2 }

TOKENS : { <id,0>, <=>, <id,1>, +, <id,2>, <*>, <id,3> }



entry index in symbol table

operators not entered in symbol table

- printf (" Total = y.d / x ", score) ;

7 tokens

Lexical Analyzer performs the foll. 2 tasks in order

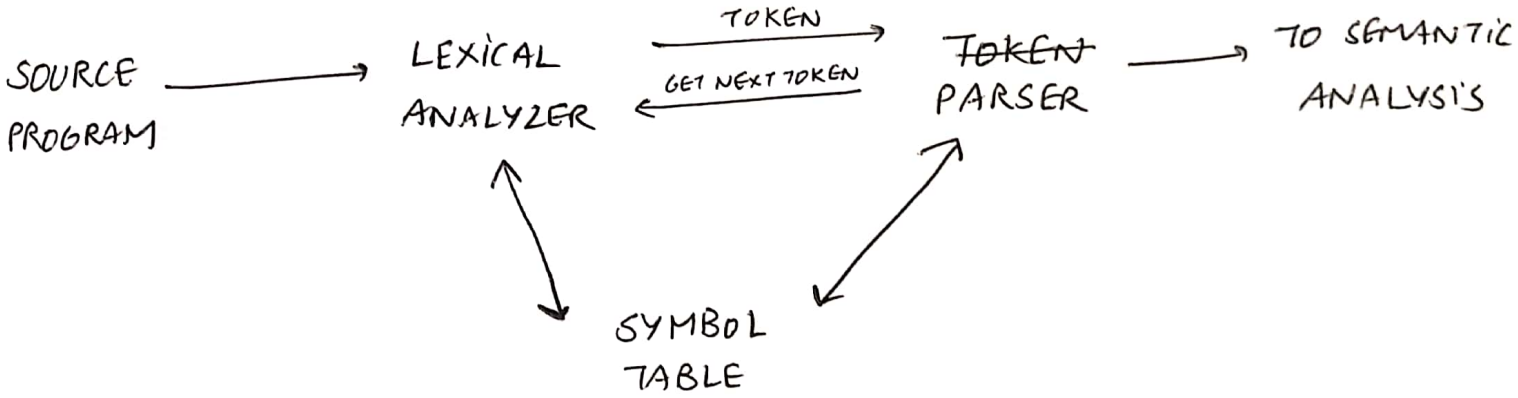
1) Scanning
2) Lexical Analysis

LEXICAL ERROR

fi (a = = x)

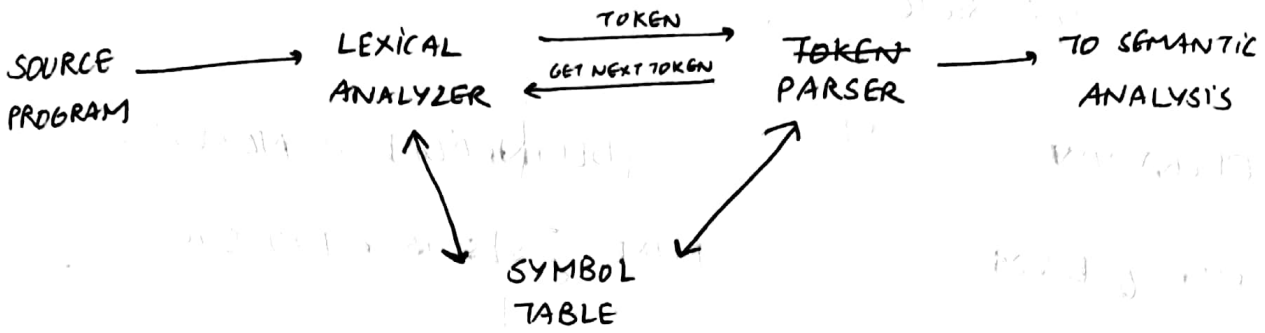
↑

considered as an identifier but later on will display error as this is a function name. lexical analyzer won't display this as an error



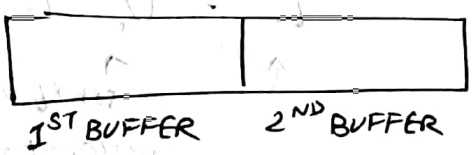
LEXICAL ERROR fi (a == x) ⟶ stored in secondary memory

↑ considered as an identifier but later on will display error as this is a function name. lexical analyzer won't display this as an error

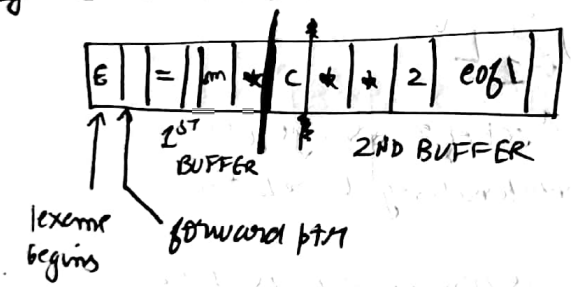


token : < token name, optional attribute >

INPUT BUFFERING ⟶ TWO Buffer scheme



Eg : E = m * C * * 2



forward ptr moves → wd character by character while lexeme begins ptr remains static

-All keywords are stored in symbol Table.

LEXICAL ANALYSIS ⟶ SPECIFICATION :- Regular expressions

↓ keyword, variable, digit recognition patterns.

LANGUAGE ⟶ alphabet → finite set of symbols eg { 0, 1 }, ASCII, UNICODE

⟶ string

↳ finite sequence of symbols drawn from the alphabet

any countable set of strings over ~~that~~ some fixed alphabets

OPERATIONS OF LANGUAGES

1) Union
2) Concatenation
3) Closure

ϵ → Empty string

OPERATION	DEFINITION & NOTATION
Union of L & M	$LUM = \{ s \mid s$ is in L or s is in M$\}$
Concatenation of L & M	$LM = \{ st \mid s$ is in L & t is in M$\}$
Kleene Closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$ (set of all strings obtained by concatenating L, zero or more times)
Positive Closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$ (set of all strings obtained by concatenating L ≥ 1 times)

A notation called Regular Expressions is commonly used for describing all the languages that can be built from these operators applied to the symbols of some alphabet

TWO RULES THAT FORM THE BASIS OF DEF^N OF REG. EXPR (RE) OVER SOME ALPHABET Σ & THE LANGUAGES THAT THOSE EXPRESSIONS DENOTE :

1) ϵ is a RE & L(ϵ) is $\{\epsilon\}$, i.e. language whose sole member is the empty string. → string whose length = 0, it's not null

2) If 'a' is a symbol in Σ, then 'a' is a RE & $L(a) = \{a\}$, i.e. the language with one string of length one.

Suppose r & s are RE denoting the languages $L(r)$ & $L(s)$ respectively then,

i) $(r) \mid (s)$ is a RE denoting the language $L(r) \cup L(s)$

ii) $(r)(s)$ is a RE denoting the language $L(r)\,L(s)$

iii) $(r)^*$ is a RE denoting $(L(r))^*$

iv) (r) is a RE denoting $L(r)$

ORDER OF PRECEDENCE : 1) * 2) concatenation 3) union } LEFT ASSOCIATIVE

EG: Let $\Sigma = \{a, b\}$

- The RE $a \mid b$ denotes the language $\{a, b\}$
- $(a \mid b)(a \mid b)$ denotes $\{aa, bb, ab, ba\}$ the language of all strings of length two over the alphabet Σ

Another RE for the same language → $aa \mid bb \mid ab \mid ba$

- a^* denotes the language consisting of all strings of ≥ 0 a's, i.e. $\{\epsilon, a, aa, aaa, \ldots\}$
- $(a \mid b)^*$ denotes the set of all strings consisting of ≥ 0 instances of a or b $\{\epsilon, a, b, aa, bb, aab, aaab \ldots\}$
- $a \mid a^*b$ denotes the language .... $\{a, ab, aaab, aab, b\}$

'b' is also a valid string → zero or more a's followed by a single 'b'

SOME ALGEBRAIC RULES THAT HOLD FOR ARBITRARY RE r, s, t

LAW	DESCRIPTION
$r \mid s = s \mid r$	'$\mid$' is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	'$\mid$' is associative
$r(st) = (rs)t$	concatenation is associative
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenation distributes over $\mid$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r \mid \epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

1) $r = \phi$, $L(r) = \{\}$, ϕ (same)
2) $r = \epsilon$, $L(r) = \{\epsilon\}$
3) $r = a$, $L(r) = \{a\}$
4) $r = a+b$, $L(r) = \{a, b\}$
5) $r = ab$, $L(r) = \{ab\}$
6) $r = a+b+c$, $L(r) = \{a, b, c\}$
7) $r = (ab+a)\cdot b$, $L(r) = \{abb, ab\}$
8) $r = a^+$, $L(r) = \{a, aa, aaa, aaa \ldots a \ldots\}$
9) $r = a^k$, $L(r) = \{\underbrace{a \ldots a}_{k \text{ times}}\}$

10) $r = (a+ba)\,(b+a)$, $L(r) = \{ab, aa, bab, baa\}$

11) $r = (a+\epsilon)(b+\phi)$, $L(r) = \{ab, b\}$

12) $r = (a+b)(a+b)$, $L(r) = \{aa, ab, ba, bb\}$

13) $r = (a+b)^*$, $L(r) = \{\phi, a, b, aa, bb, ab, ba, \ldots\}$

14) $r = (a+b)^* \cdot (a+b)$, $L(r) = \{a, b, aa, bb, ab, ba, \ldots\}$

15) $r = a^* \cdot a^*$, $L(r) = \{\epsilon, a, aa, aaa, \ldots\}$

16) $r = (ab)^*$, $L(r) = \{ab, \epsilon, abab, ababab, \ldots\}$

? : ZERO OR ONE INSTANCE eg: $L(r) \cup \{\epsilon\}$

CHARACTER CLASSES : A RE $a_1 | a_2 | \ldots | a_n \rightarrow [a_1, a_2, \ldots, a_n]$

[abc] is shorthand for a|b|c

[a-z] " " " a|b|c|...|z

Eg : C identifiers are strings of letters, digits & underscores.

Regular definition ?

letter → A|B|....|Z|a|b|....|z|_

digit → 0|1|....|9

id → letter (letter|digit)*

letter → [A-Za-z_]

digit → [0-9]

id → letter (letter|digit)*

Eg : Unsigned numbers (integers or floating pt) : are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4

Regular definition ?

digit → 0|1|....|9

digits → digit+

optionalFraction → .digits | ϵ

optionalExponent → (E (+|-|ϵ) digits | ϵ)

number → digits optionalFraction optionalExponent

digit → [0-9]

digits → digit+

number → digits (.digits)?

(E[+-]?digits)?

Q- start → if expr then start

| if expr then start else start

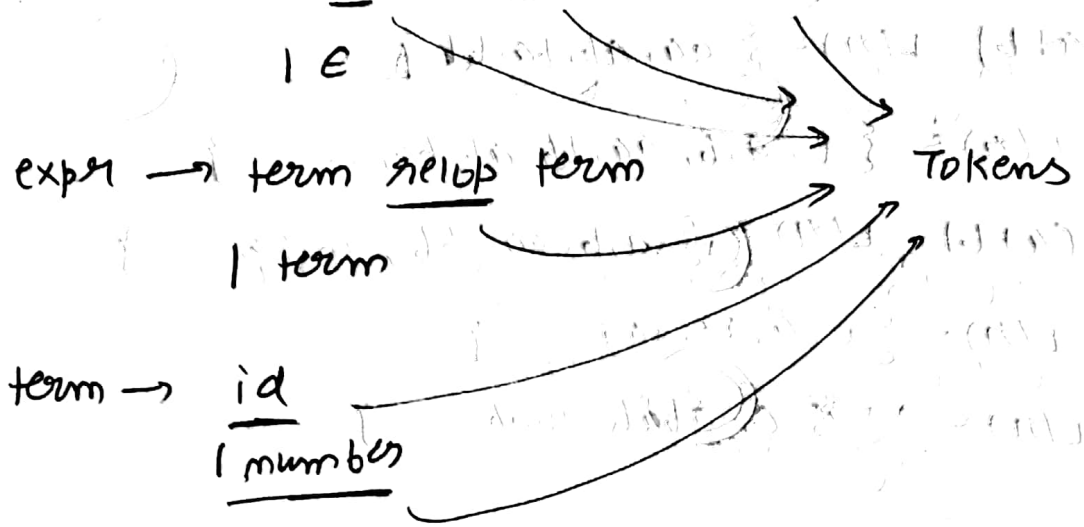
| ∈

expr → term relop term

| term

term → id

| number



Eg: ~~An ex~~ A grammar for branching statement

Patterns for branching statement

digit →

letter →

if → if

then → then

else → else

id → letter (letter | digit) *

relop → < | > | <= | >= | = | <>

ws → (blank | tab | newline) +

GOAL FOR LEXICAL ANALYZER

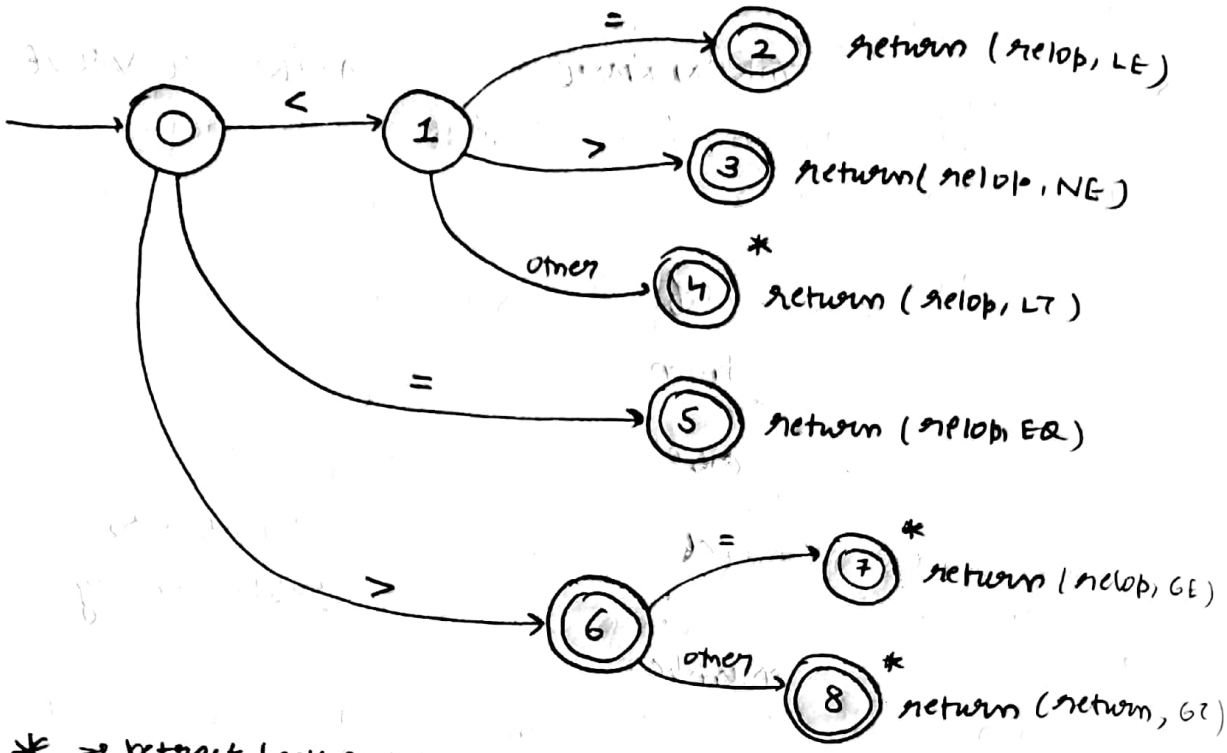
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any ws	—	—
if	if	—
then	then	—
else	else	—
Any id	id	pointer to table entry
Any number	number	" "
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagram : diff. states
↳ like a flowchart

States & edges : actions to be taken given a particular I/P

eg 

TRANSITION DIAGRAM EG



* → retract back one step (like forward ptr & lexeme begin in 2 I/P buffering, take 1 step back, till 1 back step, we'll have a token)

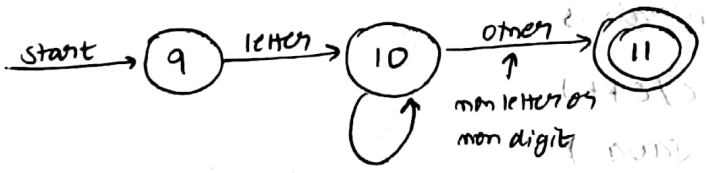
```
TOKEN getRelop()                          ← returns object of type TOKEN
{   TOKEN retToken = new(RELOP)           ← symbolic code for RELOP
    while(1)
    /* repeat character processing until a failure or return occurs */
        switch(state)
        { case 0: c = nextCharacter();
                  if (c == '<') state = 1;
                  else if (c == '=') state = 5;
                  else if (c == '>') state = 6;
                  else fail(); /* lexeme not a relop */
                  break;
          case 1: ...
          case 8: retract()
                  retToken.attribute = GT;
                  return(retToken);
        }
```

Fig Sketch of implementation of relop transition diagram

```
case 1 : c = nextchar ( );
         if ( c == '=' ) state = 2;
         else if ( c == '>' ) state = 3;
         else state = 4;
         break;
case 2 : retToken . attribute = LE;
         return (retToken);
case 3 : retToken . attribute = NE;
         return (retToken);
case 4 : retract ( );
         retToken . attribute = LT;
         return (retToken);
case 5 : ========
case 6 : ========
case 7 : ========
```

RECOGNITION OF RESERVED WORDS & IDENTIFIERS

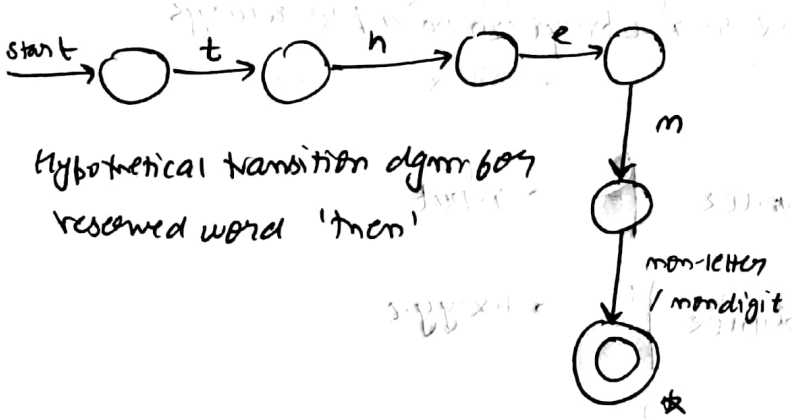
↳ 1) SYMBOL TABLE ⟶ 32 keywords in C language



return (getToken, installID())

checks in symbol table if it is a reserved word or not. In case of not a keyword, compiler identifies it as a variable & returns it token name & attribute value (pointer entry to symbol table in getToken) call.

2) TRANSITION DIAGRAM (hypothetical eg)



Hypothetical transition diagram for reserved word 'then'

CONFLICT RESOLUTION IN LEX

1) Always prefer a longer prefix to a shorter prefix

2) If longest prefix matches 2 or more patterns, prefix the pattern listed first in lex program

eg for ① ⟶ if 3 abc] 2IP lexemes

[if ⟶ if
identifier ⟶ letter (letter | digit)*

↳ here if 3 can match keyword if but acc. to longer prefix strategy, it will resolve in favour of identifier i.e. longer prefix

for ② if] 1IP lexeme

↳ matches both if & identifier but will resolve in favour of pattern listed first hence matches the keyword & conflict resolves

LEXICAL ANALYZER GENERATORS

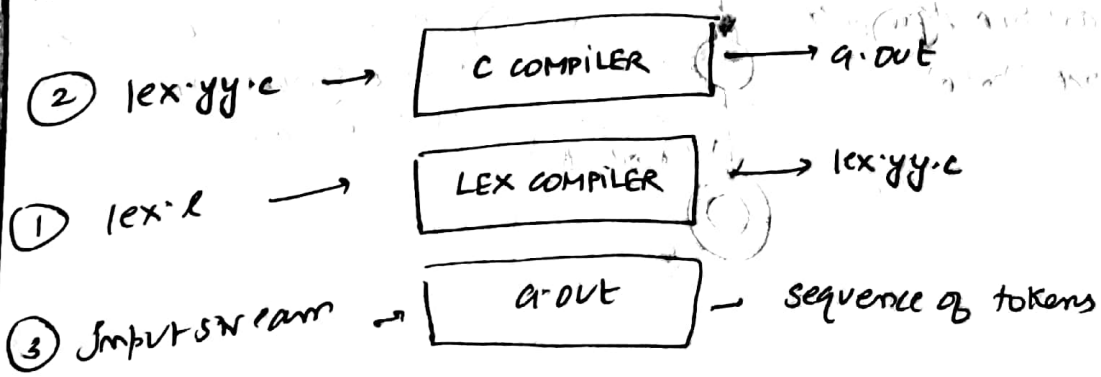
Process of generating a lexical analyzer can be automated

2 popular lexical analyzer generators:

- Flex (written in C/C++)
- Jlex (written in Java)

lex.yy.c ⟶ a file that simulates transition diagram

lex compiler transforms lex.l to a C program in a file always named lex.yy.c



STRUCTURE OF LEX PROGRAMS

declarations ← contains regex definitions (contains pattern def^n)

%.%

translat^n rules (contains (pattern, {action}) pairs)

%.%

Auxiliary functions (rest of necessary definitions)

yylval & other variables like yytext

- The program contains a %.% delimiter to mark beginning of rules & one such

PROGRAM TO DELETE FROM I/P ALL BLANKS OR TABS AT ENDS OF LINES

```
%.%
[\t]+$;
```

---

lex programs recognise only regex

Yacc writes parsers that accept a large set of CFG.

Absolute min. lex program ⟶ %.% (no def^n, no rules)

If operator characters are to be used as text characters, we must use escape '\' characters before them.

lex default action ⟶ copying I/P to O/P.

A pattern which is not matched for a string is copied as it is to O/P.

lex turns rules to a program

SYNTAX ANALYSIS

PARSING

source code → scanner → ...
↳ ...

He wrote the program

ROLE OF PARSER

1) Must distinguish b/w valid & invalid tokens
~~2) Type checking~~

Typechecking & initializat^{n} of variables is handled by semantic analyzer

Regular languages can't handle all cases

eg: strings of balanced parantheses (((---)))

∴ we require push down automata / table driven parsers.

A grammar G contains (variables (V), terminals (T), Prod^{n} Rules (P), Start (S))

Context Free Gram. CFG → why? only a single symbol is involved on LHS of → we aren't concerned with other symbols

P (Production rules)

S → aS | b → terminals
S → c → terminals

start symbol

as → as | b

not context free as 2 symbols in LHS

Derivation of a variable

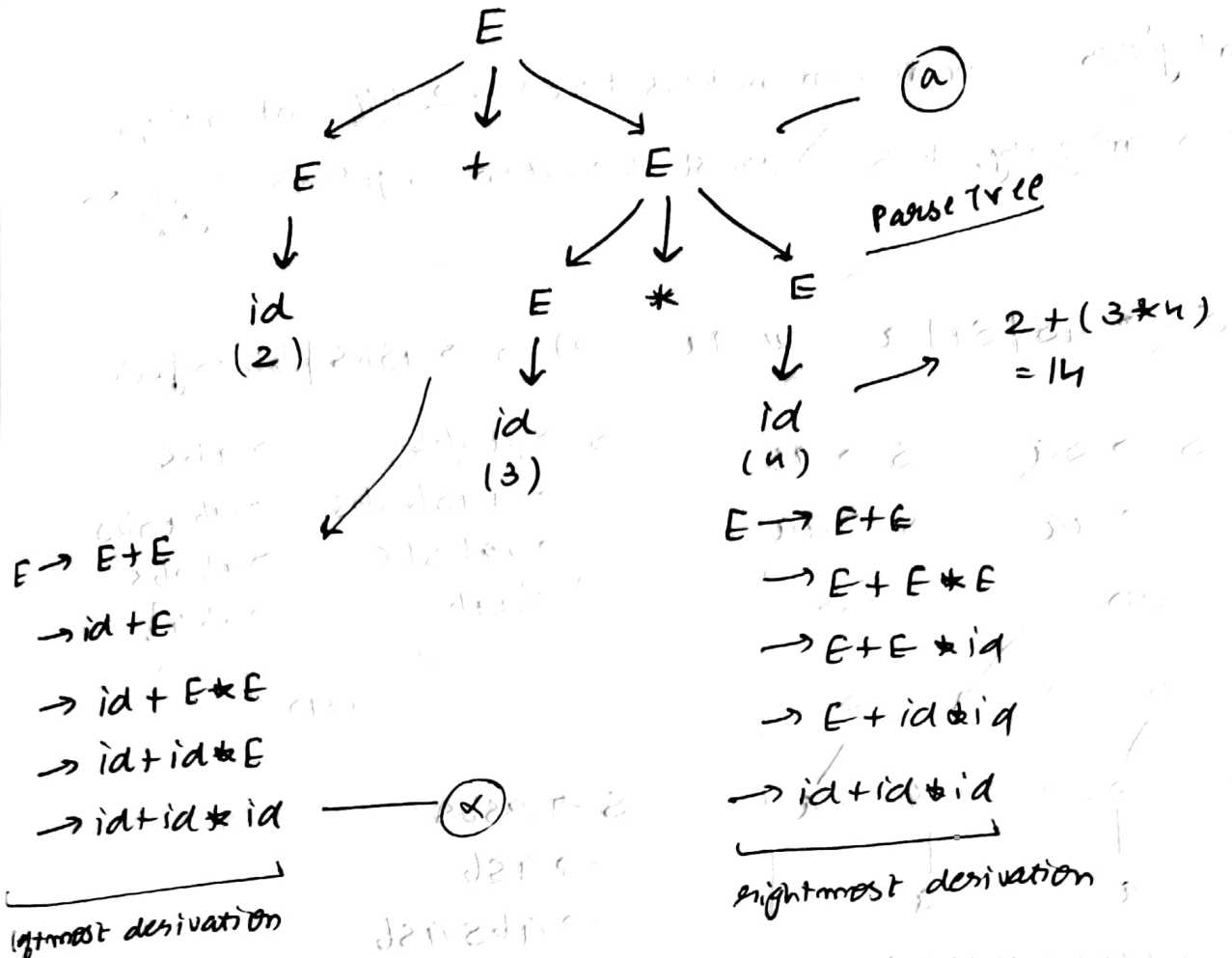
→ Leftmost Derivation ——— (1)
→ Rightmost Derivation ——— (2)

(1) we start expanding from leftmost

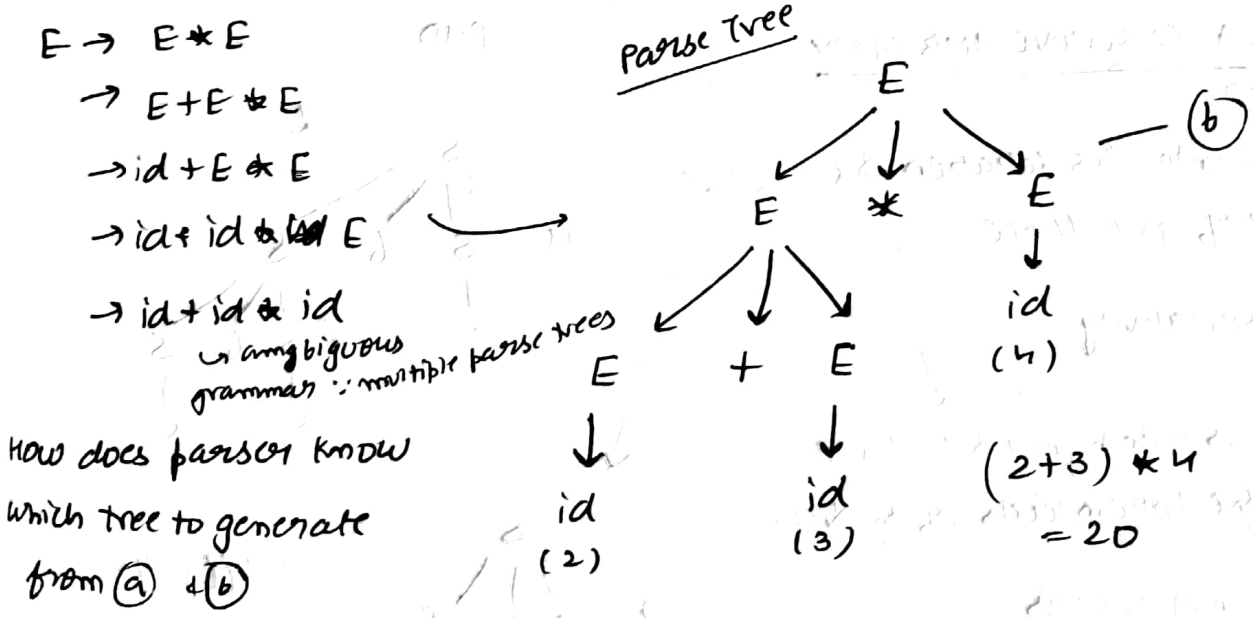
eg G : E → E+E | E*E | id

string : id+id*id

checks whether this string ∈ G or not;



Another method for deriving (α) in leftmost derivat" form



How does parser know which tree to generate from (a) & (b)

→ Ambiguity to be removed while programming.

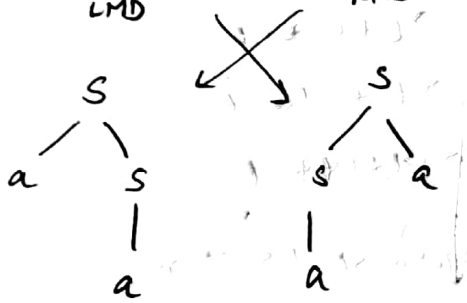
Ambiguous — more than one parse tree → 2 different interpretations

To check ambiguity, check if leftmost derivation = rightmost derivation

1) $S \rightarrow aS \mid Sa \mid a$ $\quad w = aa$

$S \rightarrow Sa \rightarrow aa$ (LMD)

$S \rightarrow aS \rightarrow aa$ (RMD)



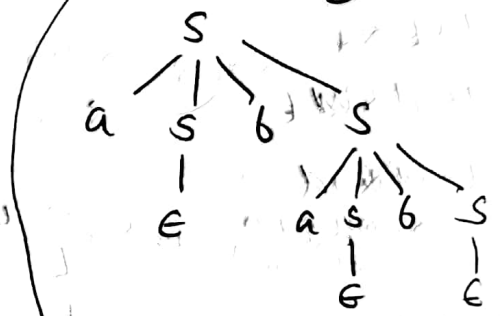
2 diff. parse trees

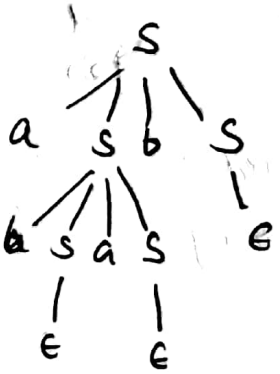
∴ ambiguous

2) $S \rightarrow aSbS \mid bSaS \mid \in$ $\quad w = abab$

$S \rightarrow aSbS$
~~$\rightarrow aaSbSbS$~~
~~$\rightarrow aabSbS$~~
~~$\rightarrow aabb$~~
$\rightarrow abS$
$\rightarrow abaSbS$
$\rightarrow ababS$
$\rightarrow abab$ (LMD)

$S \rightarrow aSbS$
$\rightarrow aSb$
$\rightarrow abSaSb$
$\rightarrow abSab$
$\rightarrow abab$ (RMD)





2 diff. parse trees

∴ ambiguous grammar

CRITERIA TO REMOVE AMBIGUITY
OR RULES

1) Take into consideration, the order of precedence

2) Associativity

operators with higher precedence lie at the bottom levels of the tree.

→ when operator is :

left associative ⇒ left recursive

right associative ⇒ right recursive

Q- Remove ambiguity from $E \rightarrow E+E \mid E*E \mid id$

($+$: 2nd, left associative; $*$: 1st, left associative)

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$ → unambiguous grammar
$F \rightarrow id$

Q- $+$ (3rd), $*$ (2nd), $\uparrow$ (power operator, 1st precedence)

$+$, $*$: left associative; $\uparrow$: right associativity

Generate a grammar which follows the above rules.

- $E \rightarrow E+T \mid T$
- $T \rightarrow T*G \mid G$
- $G \rightarrow H \uparrow G \mid id\ H$
- $H \rightarrow id$

~~$T \rightarrow F*G \mid G$~~
~~$G \rightarrow H \uparrow I \mid I$~~
~~$I \rightarrow id$~~
~~$G \rightarrow H \uparrow id \mid id$~~
~~$G \rightarrow H \uparrow I \mid I$~~
~~$I \rightarrow id$~~

PARSING TECHNIQUES / ALGOS
1) Top Down Approach
2) Bottom Up Approach

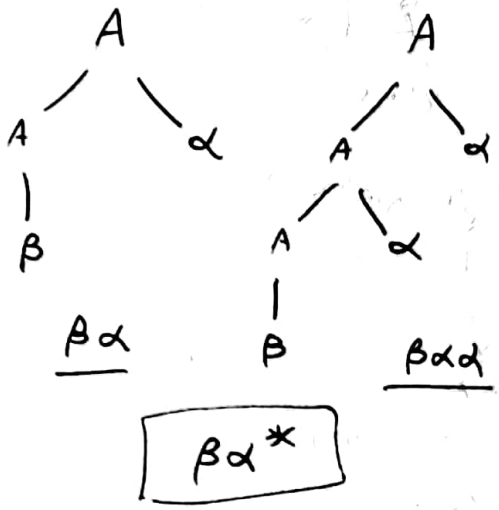
eg of Top Down Approach → Recursive Descent parsing
↓
can't handle left recursion

Elimination of Left Recursion Methods

$A \rightarrow A\alpha \mid \beta$ (left recursive)

$A \rightarrow \alpha A \mid \beta$ (right recursive)

α & β are terminals




```
A()
{
    α → stopping criterion
    A()
}
```

α*β

NO ∞ LOOP

```
A()
{
    (α) → stopping criterion
    A()
}
```

no stopping criterion here

∞ LOOP

AIM : to remove left recursion while keeping grammar same

$A \rightarrow \beta A'$

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

Grammar for $\beta\alpha^*$

$E \rightarrow iE'$

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

---

expr → expr + terminal | terminal

A A α β

left recursion eliminated

expr → terminal expr'

expr' → + terminal expr' | ϵ

① $S \rightarrow SOS1S \mid \underbrace{01}_{\beta}$

$\underbrace{S}_{A}\ \underbrace{OS1S}_{\alpha}$

first is A rest is α

$S \rightarrow \beta S'$

$S' \rightarrow \alpha S' \mid \epsilon$

② $L \rightarrow \underbrace{L,\overset{\wedge}{\$}}_{A\ \alpha} \mid \underbrace{\overset{\wedge}{\$}}_{\beta}$

$L \rightarrow \$L'$

$L' \rightarrow ,\$L' \mid \epsilon$

GRAMMAR

- Unambiguous
- Ambiguous
  - Left Recursive
  - Right recursive
    - Deterministic
    - Non-deterministic

Left factoring : converting non-deterministic to deterministic

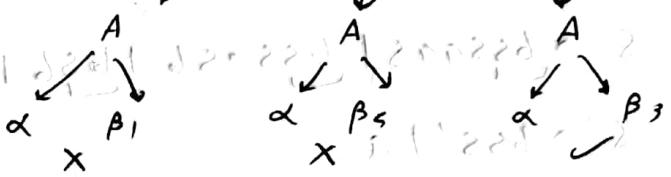
$A \rightarrow \alpha A'$

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

Deterministic

for string $w \rightarrow \alpha\beta_3$

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



backtracking problem

∴ we should try to keep grammar deterministic

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

thus non-deterministic

to make it deterministic :

$S \rightarrow aS' \mid b$

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

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

$S'' \rightarrow SbS \mid aSb$

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

non deterministic

to make it deterministic

$S \rightarrow bS' \mid a$

$S' \rightarrow SSaaS \mid SSaSb \mid Sb$

$S' \rightarrow SS''$

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

$S'' \rightarrow SS''' \mid b$

$S''' \rightarrow aaS \mid aSb$

$S''' \rightarrow aS''''$

$S'''' \rightarrow aS \mid Sb$

we can also first take bS common

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

$S \rightarrow bSS' \mid a$

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

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

$S'' \rightarrow aS \mid Sb$

TOP-DOWN APPROACH

BOTTOM-UP APPROACH

Eg: S → aABe

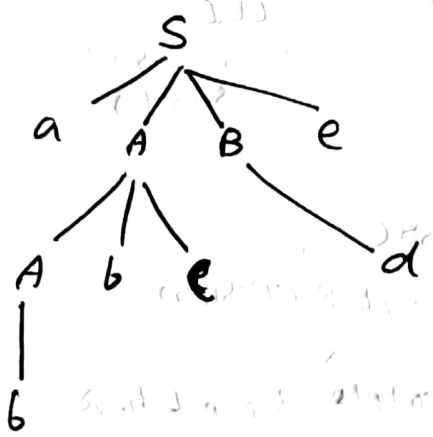
A → Abc | b

B → d

given w → abbcde

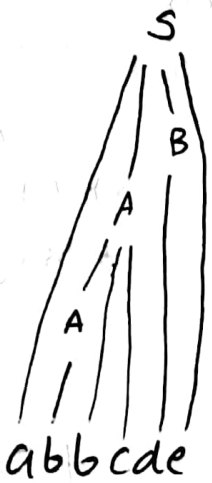
we can generate parse tree using these approaches

TOP-DOWN PARSE TREE :



Q- what production rule do we use ?

BOTTOM-UP PARSE TREE



Q- when to reduce ?

START HERE go upwards

PARSERS

- Top down approach (parsers)
  - with backtracking
    - Brute Force
  - without backtracking (should not be left recursive to use this method & must be deterministic)
    - Recursive Descent
    - LL1 (non recursive descent)
- Bottom up approach (parsers)
  - Operator precedence parsers
  - LR parsers
    - LR(0)
    - LR(1)
    - CLR(1)
    - LALR(1)

LL(1) : Deterministic (lookahead = 1)

Eliminate left Recursion

- LMD
- scanning of symbols from L to R

Structures used (components)

- Stack
- I/P buffer
- LL(1) parsing algo
- LL(1) parsing table

'$' always present in stack & buffer to keep check on when do we arrive at the end of the algo. / when to stop

data structure which helps in implementing parsing algorithms

Two parsing methodologies under LL(1)

- FIRST ( ) → first terminal before A in A
- FOLLOW ( ) → first terminal following A in prod.

LL(1) : Deterministic
Eliminate left Recursion
- LMD
- scanning of symbols from L to R

lookahead = 1

Structures used (components)
- Stack
- I/P buffer
- LL(1) parsing algo
- LL(1) parsing table

'$' always present in stack & buffer to keep check on when do we arrive at the end of the algo. / when to stop

data structure which helps in implementing parsing algorithm

Two parsing methodologies under LL(1)

FIRST ( )
↓
first terminal before A in A

FOLLOW ( )
↓
first terminal following A in proc.

S → aABCD
A → b
B → c
C → d
D → e

FIRST(S) → {a}
FIRST(A) → {b}
FIRST(B) → {c}
FIRST(C) → {d}
FIRST(D) → {e}

S → ABCD
A → b | ∈
B → c
C → d
D → e

FIRST(S) → {b, ∈, c}
" (A) → {b, ∈}
" (B) → {c}
" (C) → {d}
" (D) → {e}

$S \to ABCD$ follow(S) → $

$A \to b \mid \epsilon$ follow(A) → { c }

$B \to c$ follow(B) → { d }

$C \to d$ follow(C) → { e }

$D \to e$ follow(D) → follow(S) → { $ }

follow( ) → what is the terminal which can follow a variable in process of derivation

ϵ → should not be present , follow(S) → $

EXAMPLE

① $S \to ABCDE$

$A \to a \mid \epsilon$

$B \to b \mid \epsilon$

$C \to c$

$D \to d \mid \epsilon$

$E \to e \mid \epsilon$

	FIRST	FOLLOW
	{ a, b, c }	{ $ }
	{ a, ε }	{ b, c }
	{ b, ε }	{ c }
	{ c }	{ d, e, $ }
	{ d, ε }	{ e, $ }
	{ e, ε }	{ $ }

Q- $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

$A = \beta A'$

$A' = \alpha A' \mid \epsilon$

$E \to \underbrace{E}_{A} \underbrace{+}_{A} \underbrace{T}_{\alpha} \mid \underbrace{T}_{\beta}$

$E \to TE'$

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

$T \to \underbrace{T}_{A} \underbrace{*}_{A} \underbrace{F}_{\alpha} \mid \underbrace{F}_{\beta}$

$T \to FT'$

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

Q- $E \to TE'$

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

$T \to FT' \epsilon$

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

$F \to (E) \mid id$

	FIRST	FOLLOW
E :	{ id, (}	~~{ $ }~~ { $,) }
E' :	{ +, ε }	~~{ id, (}~~ { $ }
	{ (, id }	~~{ *, ε }~~ { *, $ }
	{ *, ε }	~~{ $ }~~ { +, $ }
	{ id, (}	~~{ *, ε }~~ { *, $ }

follow(T) = first(E') follow(T') = follow(T)

follow(E') = follow(E) follow(F) = first(T'), follow(T')

LL(1) PARSING

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

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

$A \to \beta A'$

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

↓

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

PREDICTIVE PARSING

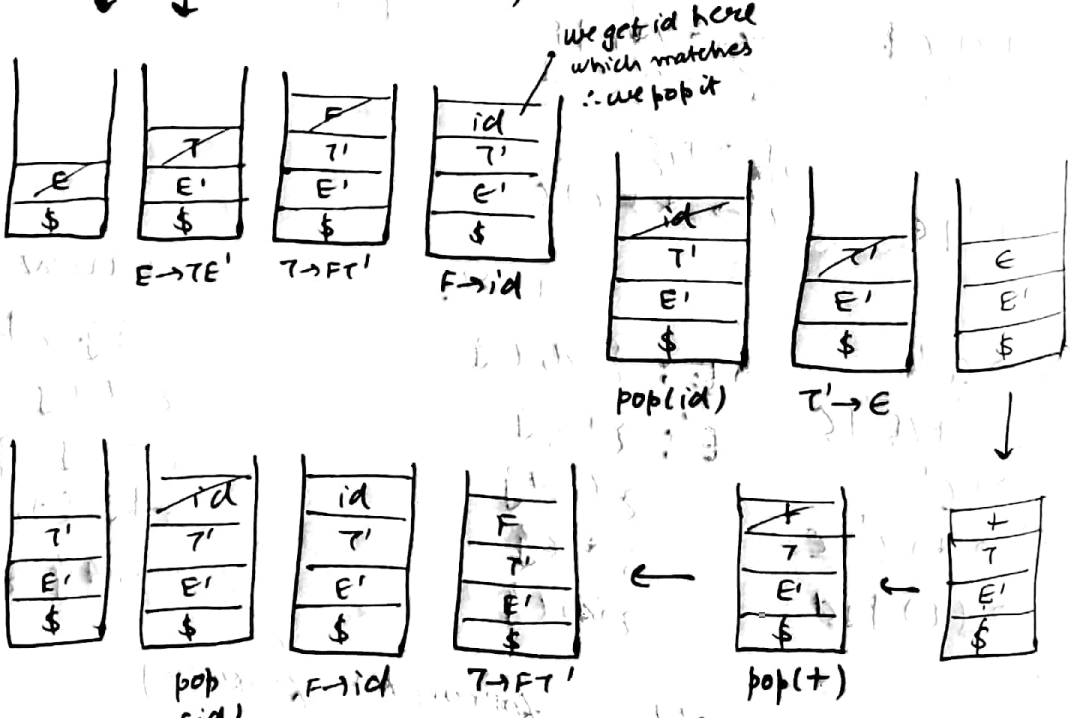
GENERATE PARSING TABLE

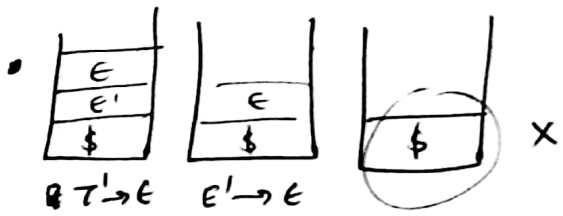
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

as $E \to +TE \mid (\epsilon)$

we use to follow(E) here

consider id | + | id | $





STACK	INPUT	ACTION
E $	id+id $	E→TE'
TE' $	↑	T→FT'
FT'E' $		F→id
idT'E' $	~~id+id $~~	pop(id)
T'E' $	id+id $	T'→ε
E' $	↑	E'→+TE'
+TE' $		pop(+)
TE' $	id+id $	T→FT'
FT'E' $	↑	F→id
idT'E' $	id+id $	pop(id)
T'E' $	↑	T'→ε
E' $	id+id $	E'→ε
$	↑	success
	$	

RECURSIVE DESCENT PARSING

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

G

function →

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

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

Functions generated for the implementation of G consisting of 2 production rules

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

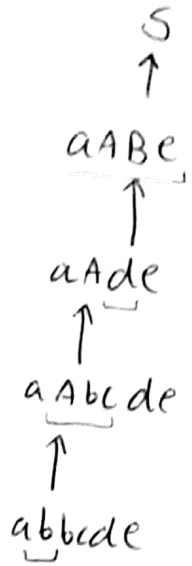
```
main()
{
  E();   ← l = getchar()  // assuming to be global
  if (l == $)
    printf("Parsing success");
}
```

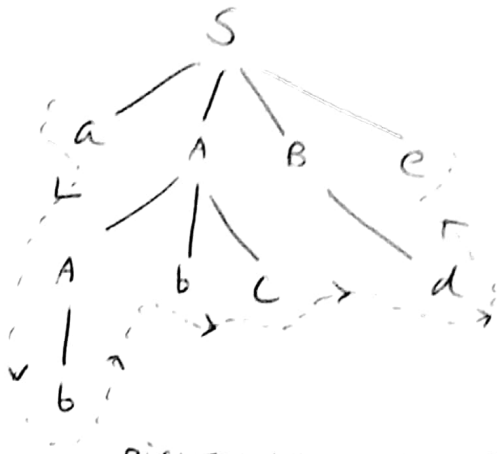
[\t] + $ { }

BOTTOM-UP PARSING (we derive the start symbol)

Eg: S → aABe
A → Abc | b
B → d

Given string, w - abbcde





RIGHTMOST DERIVATN IN REVERSE

↳ Handle: the substring that is being substituted

Handle Pruning: A handle is substring that matches body of production & whose reduction represents one step along the reverse of rightmost derivation

Eg E → E+T | T
T → T*F | F
F → (E) | id

if $w = id_1 * id_2$

Table →

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRO
$id_1 * id_2$	id_1	$F \rightarrow id$
$F * id_2$	F	$T \rightarrow F$
$T * id_2$	id_2	$F \rightarrow id$
$T * F$	$T * F$	$T \rightarrow T * F$
T	T	$E \rightarrow T$

E —— —— Parsing successful as start symbol achieved

SHIFT REDUCE PARSING : It's a form of bottom up parsing in which a stack holds grammar symbols & an input buffer holds the rest of string to be parsed. The handle always appears at the top of stack. just before it is identified as the handle.

	STACK	INPUT
Initially	$	w $
Final state	$ S	$

CONFIGURATION OF A SHIFT-REDUCE PARSER ON I/P $id_1 * id_2$

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

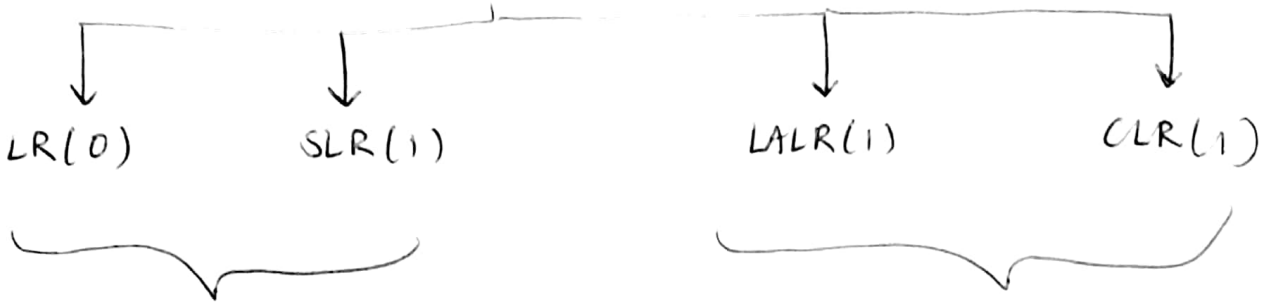
$F \rightarrow (E) \mid id$

$w = id_1 * id_2$

STACK	INPUT	ACTION
$	$id_1 * id_2$ $	shift
$ id_1	* id_2 $	reduce by F → id
$ F	* id_2 $	reduce by T → F
$ T	* id_2 $	shift
$ T *	id_2 $	shift
$ T * id_2	$	reduce by F → id
$ T * F	$	reduce by T → T * F
$ T	$	reduce by E → T
$ E	$	accept

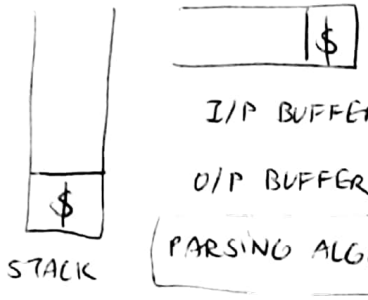
4 ACTIONS : shift, reduce, accept error

LR PARSERS



Canonical collect^{n} of LR(0) items

Canonical collection of LR(1) items



→ main difference b/w the 4 types of LR parsers

LR(0) PARSER

Eg: $S \rightarrow AA$

$A \rightarrow aA \mid b$

we include an additional product^{n} rule $S' \rightarrow S$

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

LR(0) items

$S \rightarrow \cdot AA$ item
$S \rightarrow A \cdot A$
$S \rightarrow AA \cdot$
↳ now ready for reduction

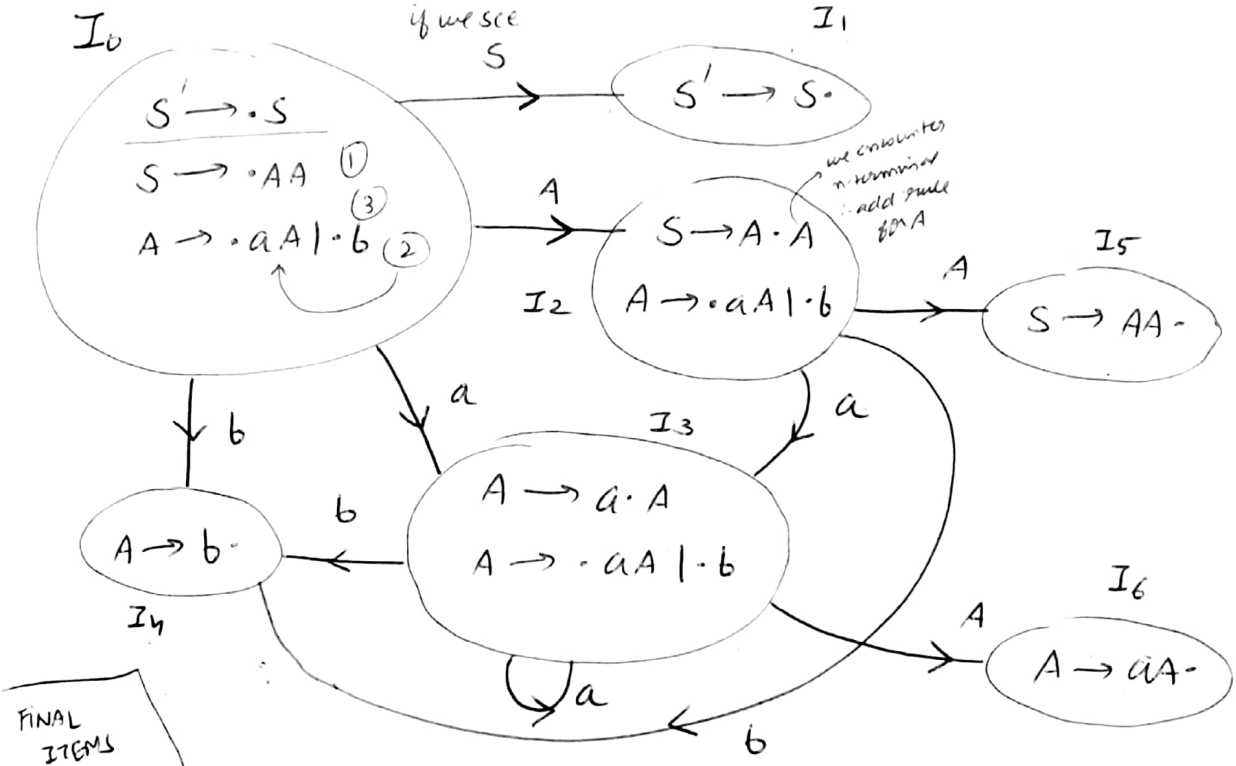
$S' \longrightarrow \cdot S$

$S \longrightarrow \cdot AA$

$A \longrightarrow \cdot aA \mid \cdot b$

adding a new production rule for encountering a n-terminal is called closure

↳ terminals ∴ no further rules



FINAL ITEMS

I_5, I_6, I_1, I_4

CANONICAL COLLECTION OF LR(0) ITEMS

	ACTION (SHIFT, REDUCE)			GO-TO	
	a	b	$	A	S
0	S_3	S_4		2	1
1			accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

LR(0) PARSING : $S' \rightarrow S$

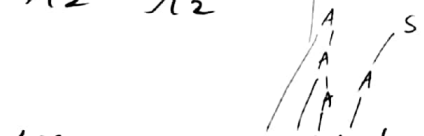
$S \rightarrow \cdot AA$ ①

$A \rightarrow \cdot aA$ ② $| \; b$ ③

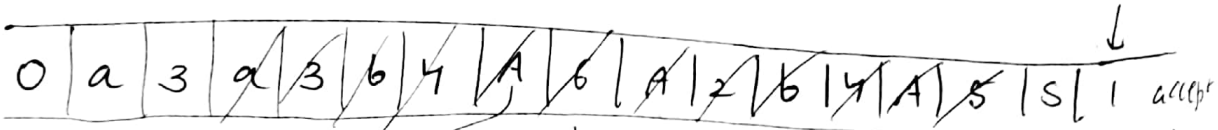
$w \rightarrow aabb\$$

Follow(A) → { a, b, $ }

	a	b	$	A	S
0	S_3	S_4		2	1
1			accept		
2	S_3	S_4		5	
3	S_3	S_4		6	
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		



for $w \rightarrow aabb\$$



parsing success

check 4, b where it is r_3 i.e. reduce

we get b through prod. no. ③ ∴ length of production = 1

∴ we pop out 2 × length of production

i.e. we pop 4 & b

pointer is at

a a b b

↑

∴ we must

r_2 ∴ length

of prod ② = 2 ∴ we pop out 4 symbols i.e. a, 3, A, b

Q- S → dA (1) | aB (2)

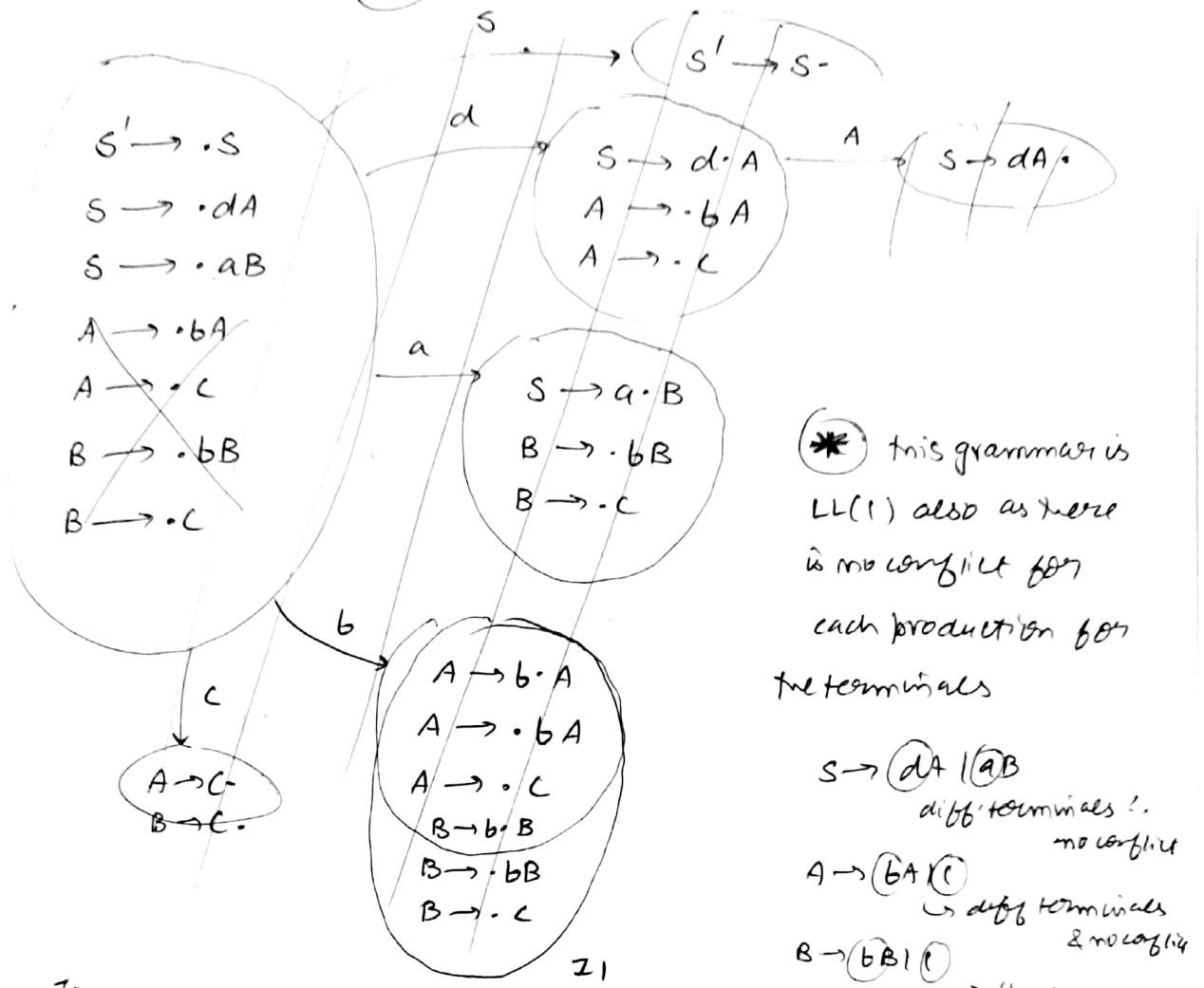
A → bA (3) | c (4)

B → bB (5) | c (6)

(*)

1) LR(0) items

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

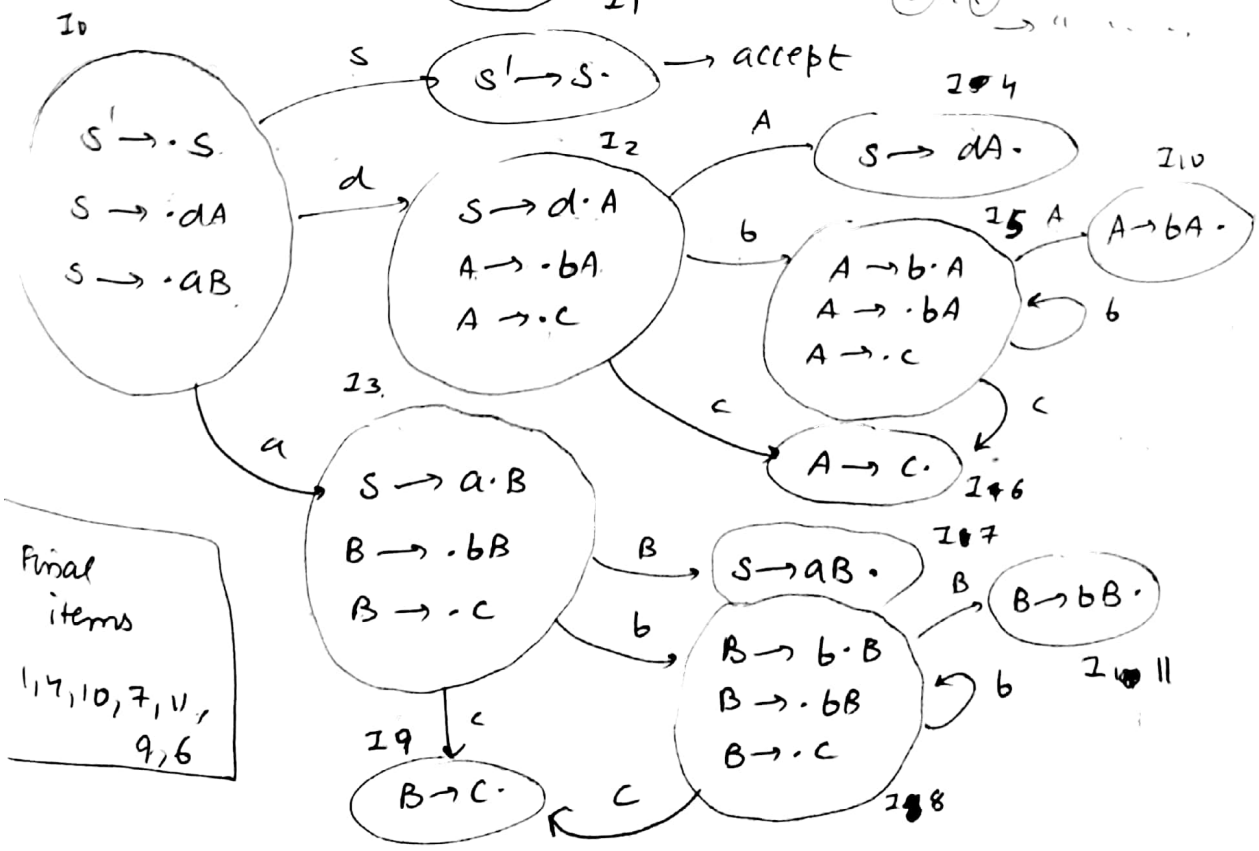


(*) this grammar is LL(1) also as there is no conflict for each production for the terminals

S → (dA) | (aB)
diff terminals ∴ no conflict

A → (bA) | (c)
↳ diff terminals & no conflict

B → (bB) | (c)
↳ " " "



Final items

I1, I4, I10, I7, I11, I9, I6

	ACTION					GO-TO	
	a	b	c	d	$	A	B
0	S_3			S_2		4	1
1					accept		
2		S_5	S_6				
3		S_1	S_9				7
4	r_1	r_1	r_1	r_1			
5			S_5	S_6		10	
6	r_4	r_4	r_4	r_4			
7	r_2	r_2	r_2	r_2			
8		S_8	S_9				11
9	r_6	r_6	r_6	r_6			
10	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5			

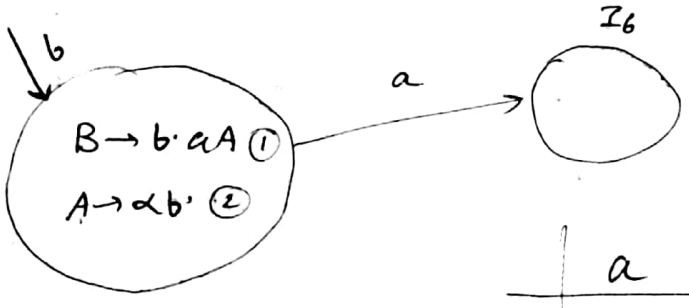
↳ this grammar is LR(0) & SLR(1)

in SLR(1), $r_1, \ldots, r_6$ is placed at the follow of LHS of production

Not every grammar is LR(0) ~~not every grammar is SLR(1)~~

POSSIBLE CASES

1) I_5



NOT LR(0)

	a	b
5	r_2/S_6	r_2

shift reduce conflict

2) I_5

① $A \to \alpha .$

② $B \to \beta .$

	a	b
5	r_1/r_2	r_1/r_2

NOT LR(0)

reduce-reduce conflict

Not every grammar is SLR(1)

POSSIBLE CASES

1) I_5 — $A \to \alpha .$ ① —a→ I_6

	a	b
5	s_6/r_1	

↳ shift reduce conflict

shift occurs if some other production also conflicts in I_5

Follow(A) = {a}

↑

shift reduce conflict

2) I_5

$A \to \alpha .$

$B \to \beta .$

when follow(A) ∩ follow(B) ≠ ϕ

↑

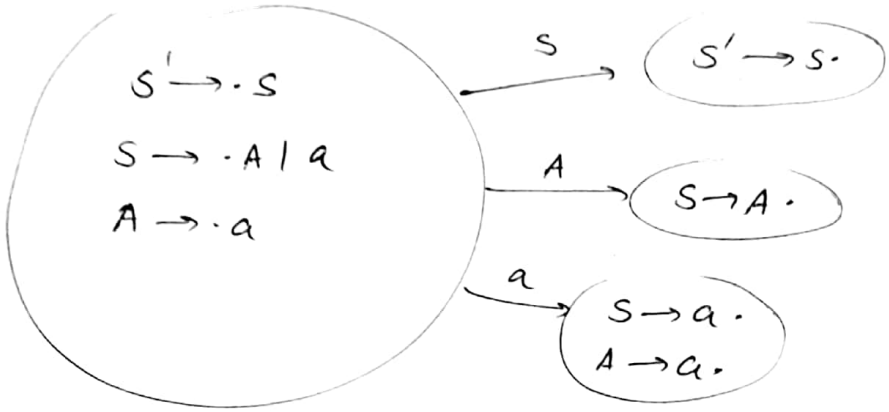
reduce-reduce conflict

	a	b
5	r_1/r_2	

Q- $S \rightarrow A \mid a$

$A \rightarrow a$

This is not LL(1) as $S \rightarrow \cdot A$ & $S \rightarrow \cdot a$ point to same terminal



case 2) of LR(0)

∴ not LR(0)

Follow(S) ∩ Follow(A) = {$} ∴ not SLR(1)

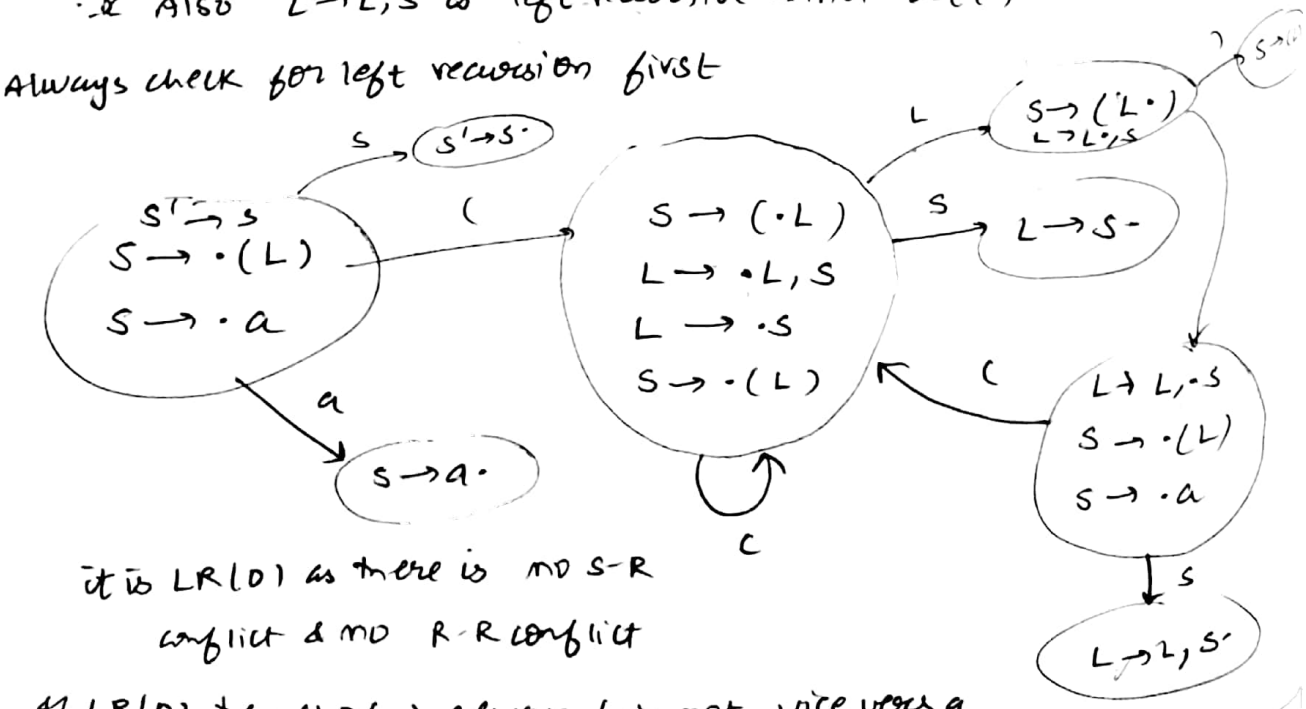
Q- $S \rightarrow (L) \mid a$ $\quad L \rightarrow S$

$L \rightarrow L, S \mid S$

This is not LL(1) as $L \rightarrow S$ & $S \rightarrow (L) \mid a$ have same terminal

∴ Also $L \rightarrow L, S$ is left recursive ∴ not LL(1)

Always check for left recursion first



it is LR(0) as there is no S-R conflict & no R-R conflict

If LR(0) then SLR(1) always but not vice versa