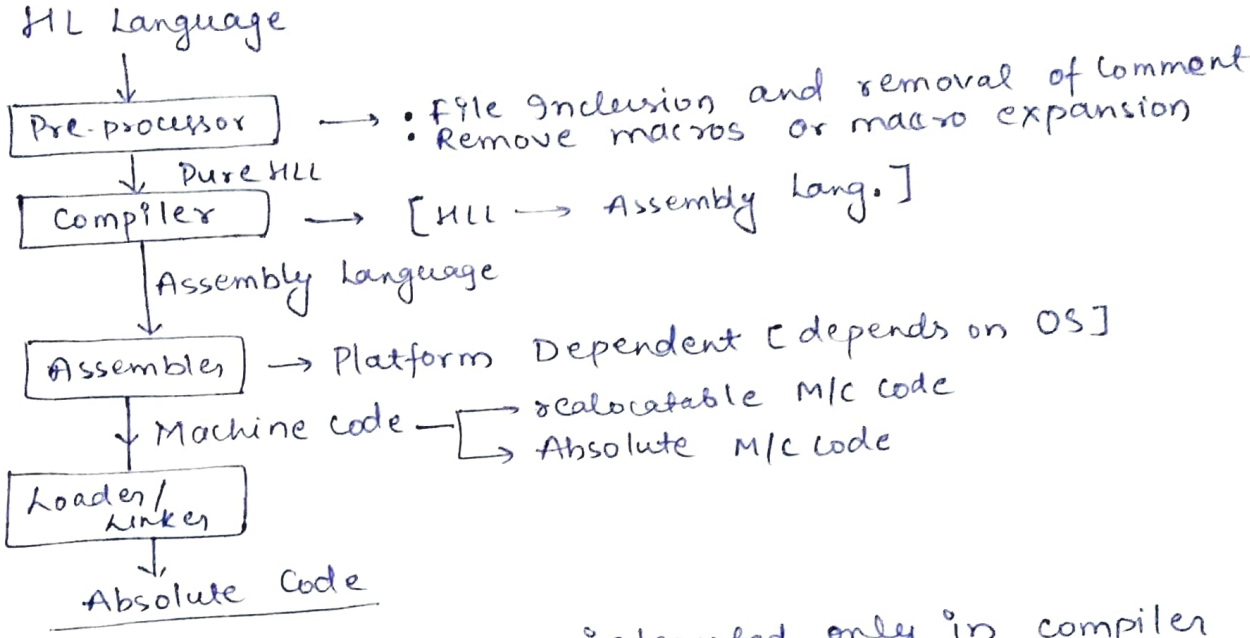
Compiler Design

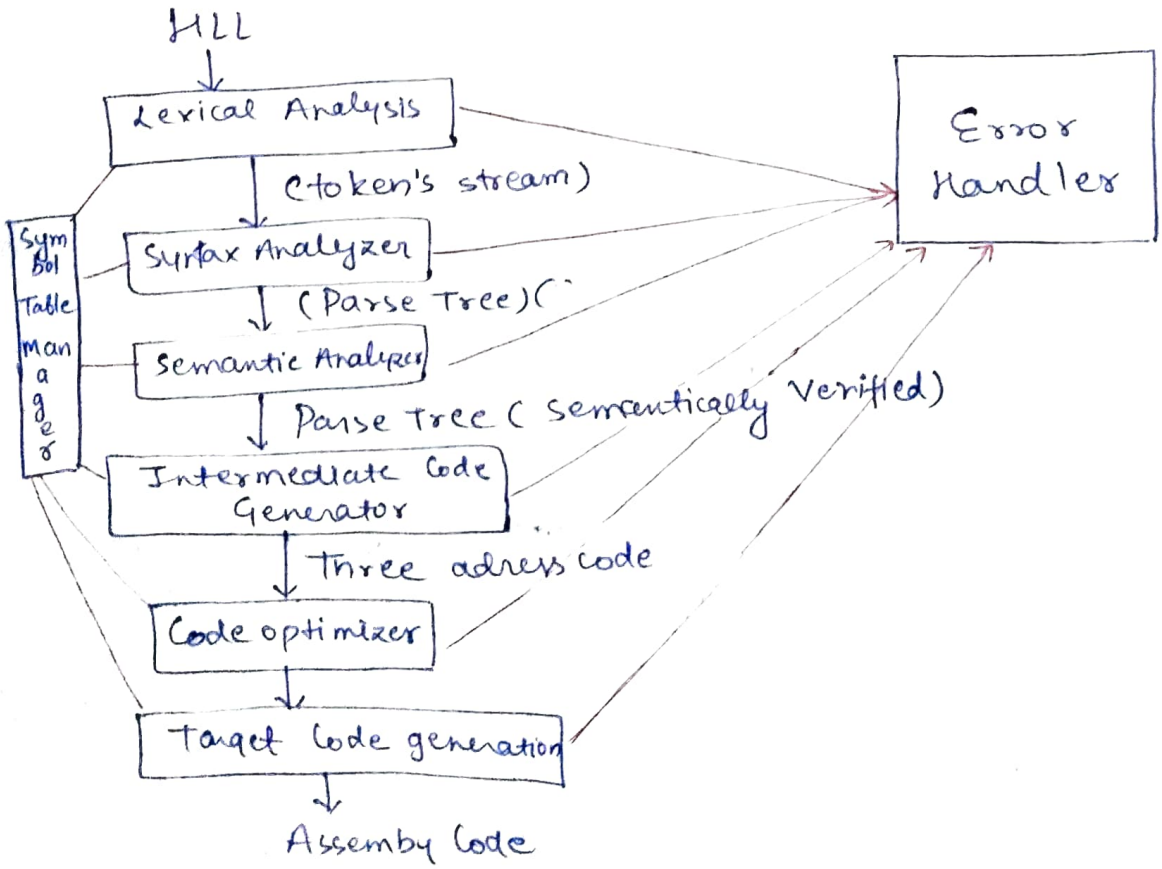
[2-5 Marks]

Introduction to various phases of Compiler

Compiler converts HL language to low-level language (0/1).



But in compiler design we are interested only in compiler phase means conversion of pure HLL to Assemby Lang.



Ex → $x = a + b * c$;

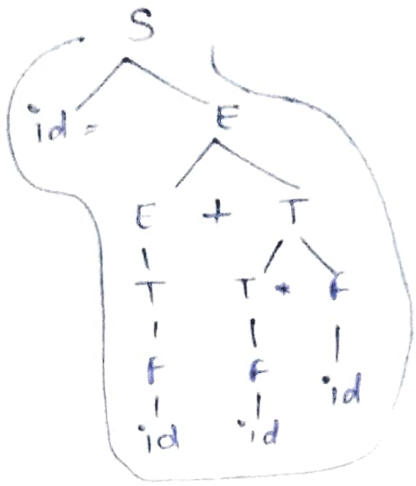
(lex) [Lexical Analyzer] → Remove white spaces

id = id + id * id → To identify a identifier we can use regular expression

(Yacc) [Syntax Analyzer] → Use set of production

Statement (S) → id = E ;
Expression (E) → E + T / T
Term (T) → T * F / F
factor (F) → id

(Parse Tree)



⇒ [Semantic Analyzer]

Parse tree semantically verified

[Intermediate code Generator]

Frontend
Backend

Three address code because it has only 3 variable required at max 3 addresses

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

[Code Optimizer]

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

[Target Code Generator]

mul R1 R2
add R0, R2
mov R2, x

Example is just to explain the working → इसका मतलब हर जगह सारे statements given हों जरूरी नहीं है।

LANCE → It provides tools to make the frontend part of compiler

'lex' is used to perform lexical analysis and Yacc is used for syntax analysis

⇒ Backend is system dependent.

Symbol Table

* It is an important data structure created and maintained by compilers in order to store information about the occurence of various entities such as variable, class, function, objects and interfaces.

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

Usage of symbol table by all the phases of compiler:

Phases	Usage
(1) Lexical Analysis	→ Creates new entries for each new identifier
(2) Syntax Analysis	→ Add information regarding attributes like type, scope, dimension, life of reference & line of use.
(3) Semantics Analysis	→ Use the available info. to check for semantics
(4) Intermediate Code Generator	→ Information in symbol table helps to add temp. variable information
(5) Code Optimization	→ Info. in symbol table used in machine-dependent optimization by considering address and aliased variable information.
(6) Target Code Generation	→ Generate the code by using the address info. of identifiers.

Symbol Table Entries

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

The attributes are –

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

→ Line of declaration contains info. of line in which variable is declared. It helps if you have an error then it will revert back the line of errors

⇒ All the attributes are not fixed size –

Limitation of fixing the size –
→ if chosen small it can't store more variables.
→ if chosen large lot of space wasted.

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

→ Line of usage helps in a way if a variable used in multiple lines.

Operations on the symbol table

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

Non-block structured: { }

* contains only one instance of the variable declaration and its scope is throughout the programme.
* for non-block structured languages operations are –
  ⇒ Insert
  ⇒ Look up

Block-structured: { { { } } } → Blocks

* variables may be re-declared and scope is within the block.
* Operations –
  ⇒ Insert ⇒ Lookup ⇒ Set (start a new one) ⇒ Reset (remove it completely)

Gate: 2012 Access time of a symbol table is logarithmic if it is implemented in:

① Linear List → Also called as linked list. In linked list implementation it will take $O(n)$.

② Search tree → Also consider as BST. So it will take $\Theta(\log_k n)$ time where $k \rightarrow$ no. of children

③ Hash Table → $O(1)$

④ None of the above → X

So, Ans → (B) Search Tree

Various Implementation on Symbol Table

Implementation	Insertion Time	Lookup Time	Disadvantages
① Linear List			⇒ Lookup time ∝ table size
(a) Ordered			
→ Array	$O(n)$	$O(\log n)$	⇒ Every insertion operation preceeded with lookup operation
→ Linked List	$O(n)$	$O(n)$	
(b) Unordered	$O(1)$	$O(n)$	
② Self Organizing List (Unsorted)	$O(1)$ [inserted at beginning]	$O(n)$	⇒ Poor performance when less frequent item searched.
③ Search tree	$O(\log_k n)$	$O(\log n^k)$	⇒ We always keep it balanced.
④ Hash table	$O(1)$	$O(1)$	⇒ When there are too many collisions T(n) may be $O(n)$

→ We consider no duplicate elements in all four Implementations.

Lexical Analyzer and Grammar

→ Phase that reads character by character and produce tokens also remove comments and white spaces.

→ Complete string is consider as a single token,

Ex "|%d|a+|b|" → not 6 tokens

↓

only one token.

Grammar = (V, T, P, S)

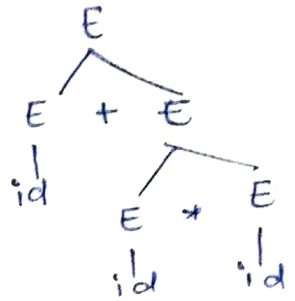
$E \longrightarrow E+E \mid E*E \mid id$

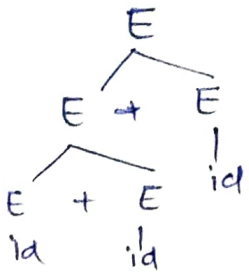
To achieve id + id * id → Left most Derivation

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

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

Parse tree





→ Since we are getting two parse tree so Grammar is ambiguous.

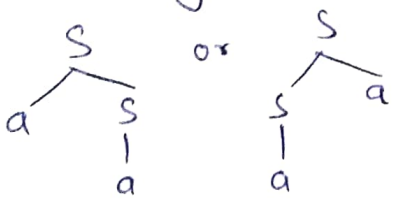
→ If a Grammar have two Right most derivation or two Left most derivation or two Parse tree then grammar is ambiguous.

There is no direct procedure for
→ To find grammar is ambiguous or not
→ To make grammar unambiguous.

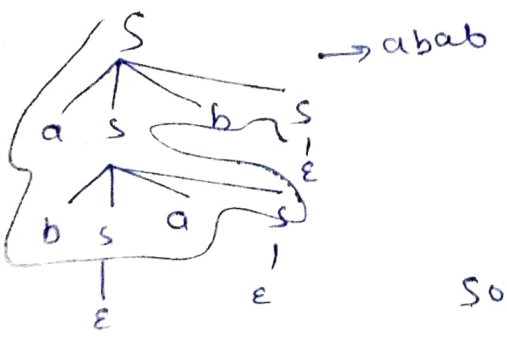
Imp → Ambiguity problem is Undecidable.

Ex ① $S \longrightarrow as \mid sa \mid a$] so ambiguous.

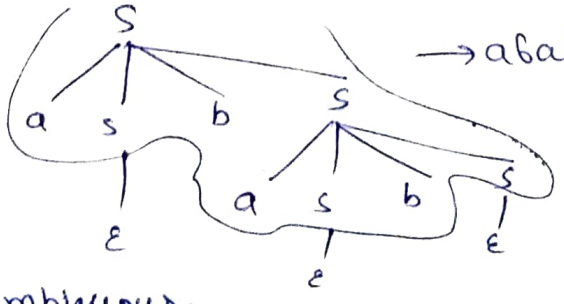
Assume a string $\omega = aa$



② $S \longrightarrow asbs \mid bsas \mid \epsilon$



→ abab



→ abab

So ambiguous.

There is no algorithm so need to it by making parse trees.

Errors and their recovery in lexical analysis

Errors: ① Numeric Literals are too long (Ex→ int A = 12345876)
② Long identifiers → longer than length given in regular expression.
③ Ill-formed numeric literals (int A = $1234)
④ Input characters are not in language.

Error Recovery:

1) **Delete:** Unknown characters are deleted (Panic mode recovery)
Ex: "Chars" corrected as "char" by deleting 's'.

2) **Insert:** Missing character is inserted to form a meaningful tokens.
Ex: "cha" corrected as "char" by inserting 'r'.

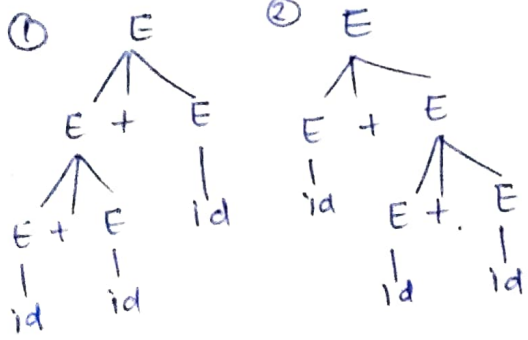
3) **Transpose:** Based on certain rules we can transpose two characters.
Ex: "whejl" can be corrected to "while"

4) **Replace:** Based on replacing one character by another.
Ex: "chrr" can be corrected as "char".

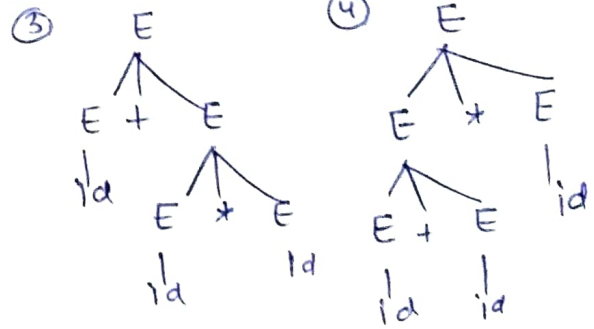
Ambiguous Grammars and making them unambiguous

Ex- $E \rightarrow E+E \mid E*E \mid E$

id + id + id



id + id * id



→ Ambiguity generates if a ~~operand~~ operator have operators on its both sides

→ In ① and ② '+' is a operator which is following left associativity rules so, ① is true and ② is false but there are two trees so grammar is ambiguous and problem is associativity.

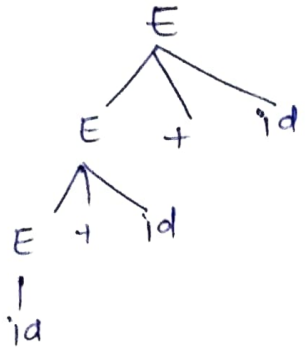
In (3) and (4) we have two operators of different precedence '+' and '*' and '*' have more precedence so '*' have to be evaluated first. But in (4) '+' evaluated first. So grammar is ambiguous with precedence issue.

In (1) and (2) we want growth in left side of tree so we need to stop the growth in right side. So grammar must be left recursive.

(Left most Symol) RHS = LHS.

$E \rightarrow E+id \mid id$

id+id+id



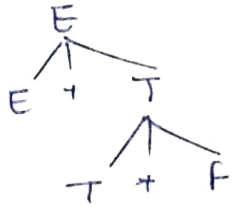
In (3) and (4) we found that highest precedence should be at lower most level.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

id+id+id



2 marks

Note: To maintain associativity use recursion(left).
To maintain precedence use levels.
To maintain power use right recursion.

Note: for power operator as ↑

Ex 2↑3↑2 → $((2)^3)^2$

Ex: $E \rightarrow E+T \mid T$ (Left recursive) → Generate first operate last

$T \rightarrow T*F \mid F$ → ——— second ——— second

$F \rightarrow G \uparrow F \mid G$ → ——— last ——— first

$G \rightarrow id$ → Right Recursive

↑ Precedence

Ex → (1) bEx → bEx or bEx | bEx and bEx | not bEx | True | false

bEx → boolean expression

E → E or F | F
F → F and G | G
G → NOT G | True | false

Unambiguous Grammar (Left Recursion)

(2) R → R+R | RR | R* | a | b | c

E → E+T | T
T → T*F | F
F → F* | a | b | c

Unambiguous

R → Regular Expression

(3) A → A $ B | B
B → B # C | C
C → C @ D | D
D → d

Precedence order $ < # < @

A → B → C → Left recursive Production so Left associativity.

(4) E → E*F | F+E | F
F → F - F | id

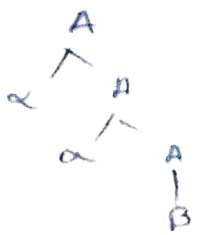
Grammar is ambiguous bcoz -
① '*' is left recursive and '+' is right recursive here
② '*' and '+' both given as same precedence.
③ '-' given more precedence over '*'.

Elimination of left recursion and left factoring the Grammars:

Recursion

Left Recursion
A → Aα | β
Language ← [βα*]

Right Recursion
A → αA | β
[α*β] → language



TOP-DOWN Parsers doesn't operate on Left-Recursion.
So we try to remove Left Recursive Grammars.

Given $A \rightarrow \beta\alpha^+$ but we want to remove α^+

So, Soln

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

→ Right recursive Grammar equivalent to Left recursive Grammar

(2) $E \rightarrow E+T \mid T$ ($A \rightarrow A\alpha + \beta$ form)

A : E ; α : $+T$; β : T

$(A \rightarrow \beta A')$: $E \rightarrow TE'$

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

(3) $S \rightarrow S0S1S \mid 01$

A : S ; α : $0S1S$; β : 01

Soln $A \rightarrow \beta A'$: $S \rightarrow 01S'$

$S' \rightarrow \epsilon \mid 0S1SS'$

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

$L \rightarrow L,S \mid S$

A : L ; α : $,S$; β : S

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

$\approx$

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

(5) $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots \mid \beta_1 \mid \beta_2 \mid \cdots$

Soln

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \cdots$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots \mid \epsilon$$

Grammars type

(1) Ambiguous and Unambiguous

(2) Left Recursive and Right Recursive.

We don't want ambiguous and Left Recursive grammar.

③ Deterministic and non-deterministic Grammar

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ (Multiple option)

Problem → Backtracking not Allowed

Common prefix problem → Since each element ($\alpha\beta_1, \alpha\beta_2, \alpha\beta_3$) have '$\alpha$' as prefix. So it may create ambiguity as-

Suppose input is $\alpha\beta_3$ then, parser have to decide which have to choose from ($\alpha\beta_1, \alpha\beta_2$ and $\alpha\beta_3$) just by looking α. Suppose it choose $\alpha\beta_2$ and we have $\alpha\beta_3$ in input then it needs to backtrack and it is not possible.

Soln

$A \to \alpha A'$

$A' \to \beta_1 \mid \beta_2 \mid \beta_3$] Left Prefixing or factoring. (Deterministic)

→ Conversion of Non-deterministic to Deterministic called 'left factoring'.

Ex ① $S \to iEtS \mid iEtSeS \mid a$

$E \to b$

Soln

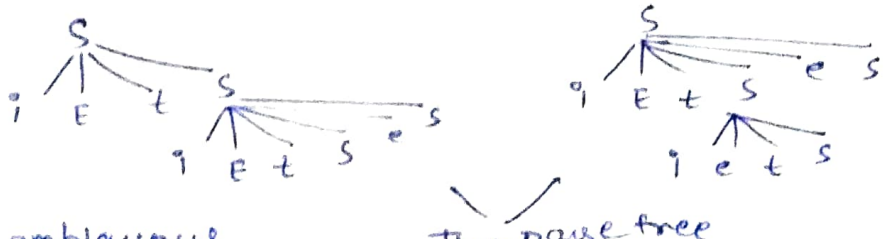
$S \to iEtSS' \mid a$

$S' \to \epsilon \mid eS$

$E \to b$

Check for ambiguity

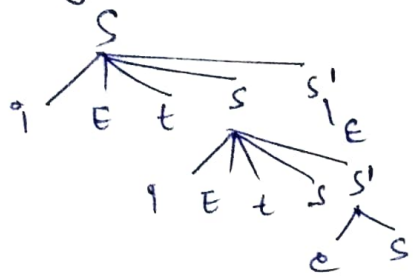
Take a string: iEtiEtSeS (Intermediate string of parse tree)



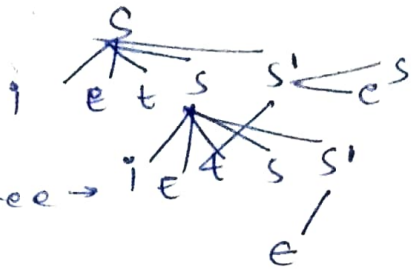
two parse tree

→ So, grammar is ambiguous before Left factoring

After left factoring



← 2 Parse tree →



→ Eliminating non-determinism doesn't guaranteed ambiguity removing.

→ If we have if statement more than else statement then it cause a problem called dangling-else problem.

Soln → Always associate else with its nearest if.

Remove Non determinism

Ex- ① $S \rightarrow assbs \mid asasb \mid abb \mid b$

either pull 'a' as common prefix from all '3'

Pull 'as' as prefix from start 2

$S \rightarrow ass' \mid abb \mid b$

$S' \rightarrow sbs \mid asb$

① $S \rightarrow as' \mid b$

$S' \rightarrow ssbs \mid sasb \mid bb$

③ steps in soln

② $S' \rightarrow ss''$

③ $S'' \rightarrow sbs \mid asb$

Ex ② $S \rightarrow bssaas \mid bssasb \mid bsb \mid a$

Soln $S \rightarrow bss' \mid a$

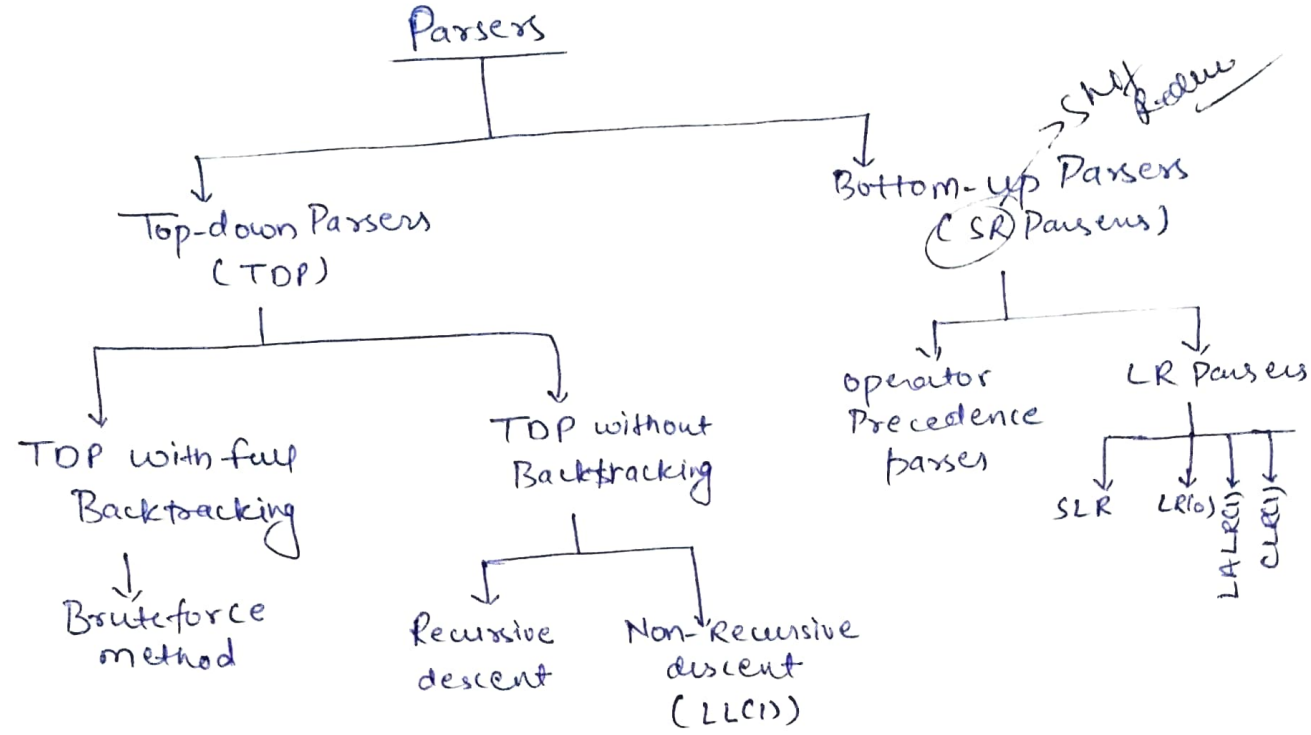
$S' \rightarrow saas \mid sasb \mid b$: $S' \rightarrow sas'' \mid b$

$S'' \rightarrow as \mid sb$

Reduces to

Parsers (Syntax Analyzer)

Introduction to Parsers, LL(1) Parsing

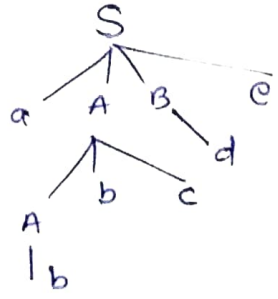


→ TDP without backtracking can be done by removing left recursion (To accept top-down necessary) and non-determinism. (for non backtracking)

→ 'operator precedence parser' is only parser that can accept ambiguous grammar.'

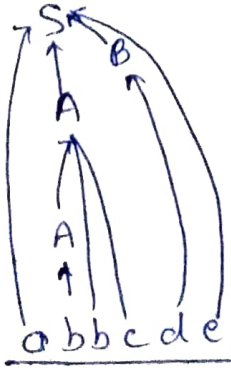
Parsers task is to verify that given string can be generated or not using parse tree.

Ex → S → aABe
A → Abc | b
B → d



Top to down selection of production.
what to use?

String(w) = abbcde



→ when to reduce?
→ what to use?

→ Bottom-to-up approach

Top down parser

$S \Rightarrow aABe$
$\Rightarrow aAbcBe$
$\Rightarrow abbcBe$
$\Rightarrow abbcde$

→ Way to acheive the paused string is Left most derivation

Bottom up parser

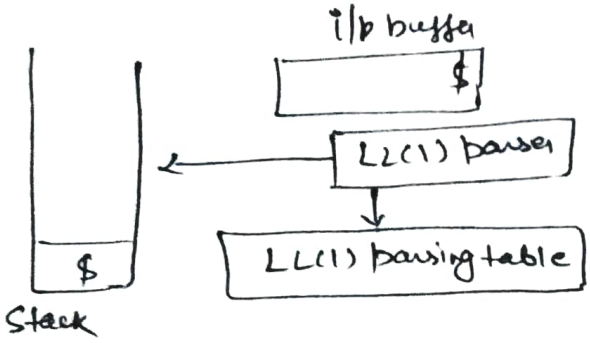
$S \Rightarrow aABc$
$\Rightarrow aAde$
$\Rightarrow aAbcde$
$\Rightarrow abbcde$

→ Way to acheive bottom-up parsing is right most derivation in reverse.

LL(1) → no of look aheads
- from Left to right
- Left most derivation

Components of Parser

1. LL(1) parser → Contain parsing algorithm
2. LL(1) parsing table → Construct using Grammar
3. Stack → used for parsing.
4. i/p buffer → inputs with $ at bottom of stack



① First()

$S \rightarrow aABC$
$A \rightarrow b$
$B \rightarrow c$
$C \rightarrow d$

First(S) → a
(A) → b
(B) → c
(C) → a

first element in string.

Ex: $S \rightarrow ABC$
$A \rightarrow b \mid \epsilon$
$B \rightarrow c$
$C \rightarrow d$

Here first(S) will be derived from A if A derived 'b' then first(S) is b otherwise first(S) will be the value derived by the B.

② Follow()

$S \rightarrow ABCD$
$A \rightarrow b \mid \epsilon$
$B \rightarrow c$
$D \rightarrow e$
$C \rightarrow d$

follow(S) → look RHS of each production if anyone have then do the operation performed for follow(A) otherwise

follow(S) → { $ } → Dollar only

$ follows each starting symbol.

⇒ follow(A) : S → A B C D contains A so follow(A) will be the first terminal of complete RHS.

$\frac{A}{LHS}$ $\frac{B\,C\,D}{RHS}$ → first variable is B that always generated ε

⇒ So follow(A) → {c}

⇒ Similarly follow(B) → $\frac{B}{LHS}$ $\frac{C\,D}{RHS}$ = {d}

⇒ follow(C) → $\frac{C}{}$ $\frac{D}{e}$: {e}

⇒ follow(D) → if it is not follow by any variable then include $.

follow(D) = {$}

Short Note

'ε' can't be included in follow() sets but first() can contain 'ε'. It is not necessary that '$' must come in each follow().

Ex. on first and follow in LL(1):

		first()	follow()	
①	S → ABCDE	{a, b, c}	{$}	→ Grammar must not be left recursive.
	A → a/ε	{a, ε}	{b, c}	
	B → b/ε	{b, ε}	{c}	
	C → c	{c}	{d, e, $}	
	D → d/ε	{a, ε}	{e, $}	
	E → e/ε	{e, ε}	{$}	
②	S → Bb/Cd	{a, b, c, d}	{$}	
	B → aB/ε	{a, ε}	{b}	
	C → cC/ε	{c, ε}	{d}	

		first()	follow()
③	$E \to TE'$	{id, (}	{$,)}
	$E' \to +TE' \mid \epsilon$	{+, ε}	{$,)}
	$T \to FT'$	{id, (}	{+, $,)}
	$T' \to *FT' \mid \epsilon$	{*, ε}	{+, $,)}
	$F \to id \mid (E)$	{id, (}	{*, +, $,)}
④	$S \to ACB \mid CbB \mid Ba$	{d, g, h, ε, b, a}	{$}
	$A \to da \mid BC$	{d, g, h, ε}	{h, g, $}
	$B \to g \mid \epsilon$	{g, ε}	{$, a, h, g}
	$C \to h \mid \epsilon$	{h, ε}	{g, $, h, b}
⑤	$S \to aBDh$	{a}	{$}
	$B \to cC$	{c}	{g, f, h}
	$C \to bC \mid \epsilon$	{b, ε}	{g, f, h}
	$D \to EF$	{g, f, ε}	{h}
	$E \to g \mid \epsilon$	{g, ε}	{f, h}
	$F \to f \mid \epsilon$	{f, ε}	{h}

if follow(X) = ε then follow(LHS) = follow(RHS)

Construction of LL(1) parsing table

We are constructing the parsing table of set of productions ③ above

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

(Terminals) →

(Variable) ↑

If RHS of production contain ε then place the production in the follow() of LHS.

⊗ id

E | $E \to TE'$ means we can achieve id from E using $E \to TE'$, and place in first(T).

Ex-(2) $S \rightarrow (S) \mid \epsilon$

first(S) → { (, ϵ }

follow(S) → {), $ }

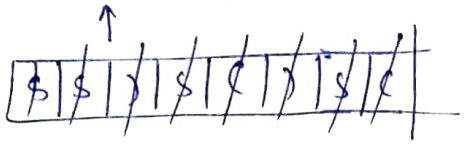
	(	)	$
S	S→(S)	S→ϵ	S→ϵ

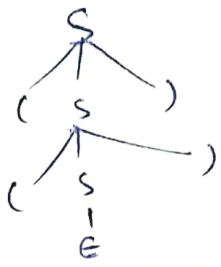
for first(S) — for ϵ

Grammar used for generating balanced parantheses

(S) → ((S)) — — (((--)))

Ex → ω = (( )) $





Top-down parser will not work if a single column in a table contains multiple productions, for each row.

Disadvantage of LL(1)

(3) $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

	a	b	$
S	S→AaAb	S→BbBa	
A	A→ϵ	A→ϵ	
B	B→ϵ	B→ϵ	

gmb

for LL(1) parsing grammar must not be left recursive or non-deterministic.

(4) $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid - - - \alpha_n$

	a	b	c	d	. . .
A					

This gramar is deterministic and also not left recursive but not the suitable for LL(1) parsing because containing more than one value in same colum.

→ it can be accepted if they are mutually exclusive.

(i) $S \rightarrow asbS \mid bSaS \mid \epsilon$

	a	b	$
S	S→asbS S→ϵ (X)	S→bSaS S→ϵ (X)	S→ϵ

not possible and it is ambiguous.

(ii) $S \rightarrow aABb$

$A \rightarrow c \mid \epsilon$

$B \rightarrow d \mid \epsilon$

Problem may arises if there is an alternative as -

A → can derive c or ϵ both.

→ No ambiguous grammar is LL(1).

$S \rightarrow aB/\epsilon$
$B \rightarrow bC/\epsilon$
$C \rightarrow cS/\epsilon$

	a	b	c	$
S	S→aB			S→~~εB~~
B		B→bC		B→ε
C			C→cS	C→ε

↳ Grammar is LL(1).

Recursive descent parser (RDP)

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

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

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

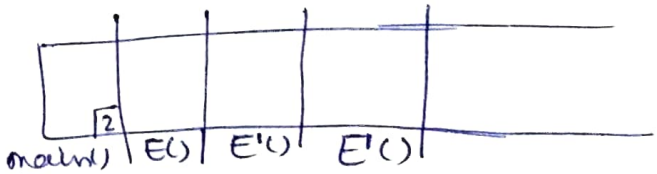
] for ε

```
match(char t)
{ if(l=='t')
   l=getchar(); ] → to increment 'l'
  else
   printf("error")
]
```

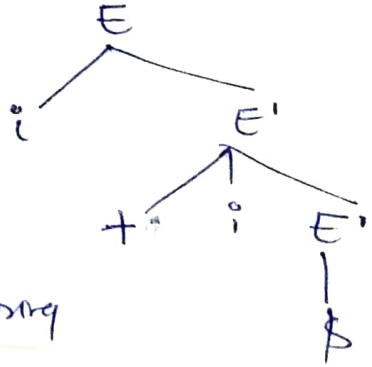
l → look ahead pointer variable

```
main(){
1.   E();
2.  if(l=='$')
3.   print("successful")
  }
```

Ex i+i$ to generate



↓
Recursion Stack



→ In RDP we make recursive functions for every variable (LHS) and then try to parse the string (Input) using functions.

→ Normal stack used for recursion explained in activation record concept

Operator grammar and operator precedence parser
(Bottom-UP Parser)

Operator Grammar

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

In operator grammar no two operands are adjacent means operator used as separator without 'ϵ' production.

$E \rightarrow EAE$
$A \rightarrow \epsilon \mid + \mid *$
] not an operator grammar becaus EA two variables are adjacent.

$S \rightarrow SbSbS \mid SbS \mid a$] → Operator Grammar

Imp

Operator precedence parser can even parsed the ambiguous grammars

Operator Relation Table

$E \rightarrow E+E \mid E*E \mid id$ → ambiguous.

	id	+	*	$
id	x	>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	x

(Terminals)

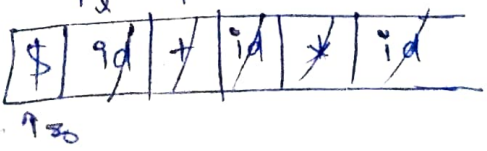
* Two id together means two variables together which is not possible (x).
* id have highest precedence over all operators
* $ have least precedence

⇒ '+', '*' have left associativity. So ~~left~~ in comparison of two same operators decision should be taken on the basis of precedence / associativity.

Left(+) > Right(+).

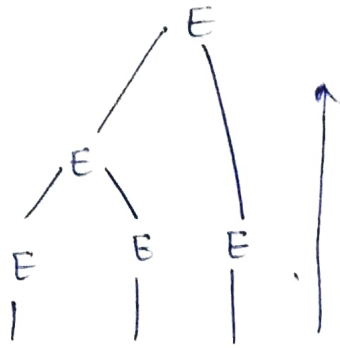
Ex → id + id * id $

E E E



if ($i > z_0$) push (i) else pop (z_0)

Reason → Pop(id) mean reduce it to variable as LHS as we doing in Bottom up Parser.

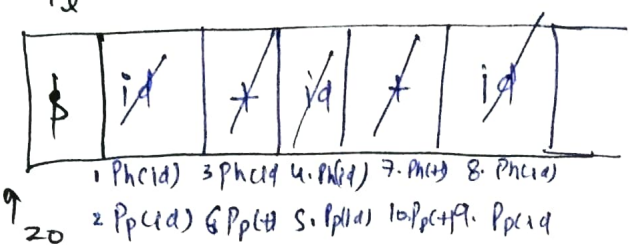


Bottom up →

Even the grammar is ambiguous but we get just a single parse tree due to precedence rule defined in table.

② id + id * id $

Ph → Push( )
Pp → Pop( )

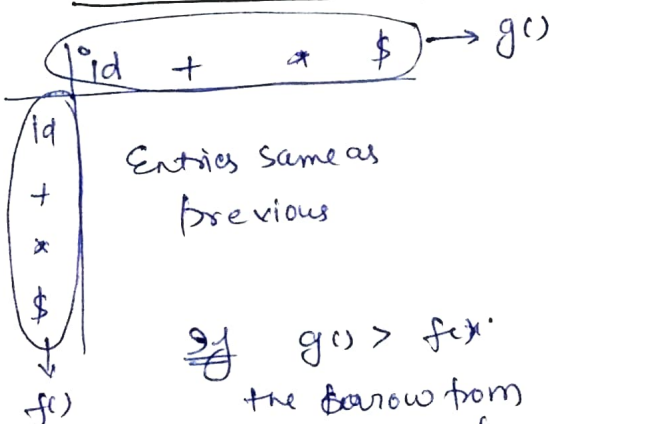


Note → open brace '(' and close brace ')' will get same (=) precedence

Disadvantage of table: size → $O(n^2)$ ∀ n → no. of operator

To reduce the size we go for,

Operator function Table

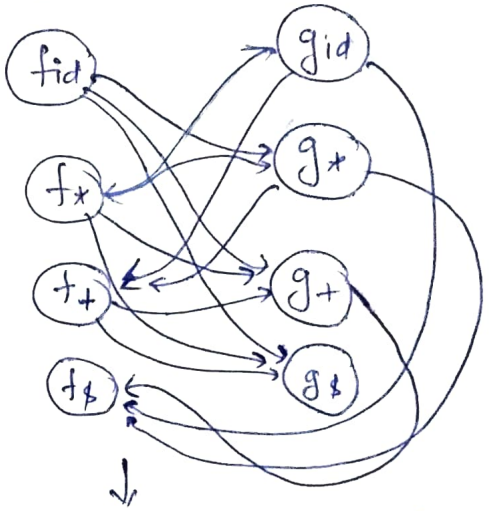


Entries same as previous

If g( ) > f*
the arrow from g → f

Ex gid > f+
So → from gid → f*

function nodes



Graph has no cycle
If there is any cycle it means you can't construct the function table

Longest path from (i) fid

fid → g* → f+ → g+ → f$

(ii) **from gid**

gid → f* → g* → f+ → g+ → f$

After finding longest path from each node than Calculate no. of node in the path and insert (n-1)

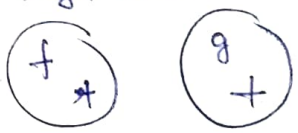
→ To see

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

function table

O(2n)

f node are left side and g node are right side



4 > 1 → * > +

This way you can take decisions.

Disadvange of function table :

Blank entries in operator relation table is converted to entries in function table

error

→ Error detecting capability of function table is less than relation table.

→ But we generally use function table.

Ex

P → SR | S
R → bSR | bS
S → wbS | w
w → L*w | L
L → id.

not an operator Grammar

Put R in P

P → SbSR | SbS | S

P → SbP | SbS | S

× R → not recursive now

S → same as previous.

w →

L →

Operator reln table

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

⇒ L*w is right recursive so star(*) at right will get higher precedence over left (*).

⇒ Precedence can be decided according to level of occuring of symbol.