6th January, 2025

Compiler Design

→ Self hosting Compiler : compiling using same language
eg: C, C++, C#, Java

↓ process

bootstrapping - technique for producing a self-compiling compiler.

→ Native Compiler : generates code for same platform
eg: GCC

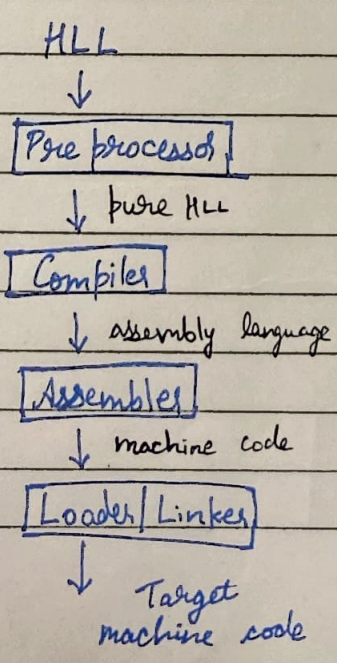
→ Cross Compiler : direct compilation not possible ⇒
build in build env, use in target env
eg: embedded systems (washing machine)

→ Transpiler : Source-to-Source compiler
(HL → HL)

→ Decompiler : low level → high level

→ Compiler-compiler : tools to generate compiler

Language Processing System



Preprocessed : * Remove comments
* copy header files
* expand macros

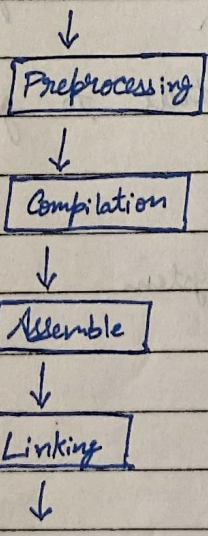
target code → assembly code

Optimization phase : Optional

Scanner = Lexical analysis
↓
Valid tokens ?

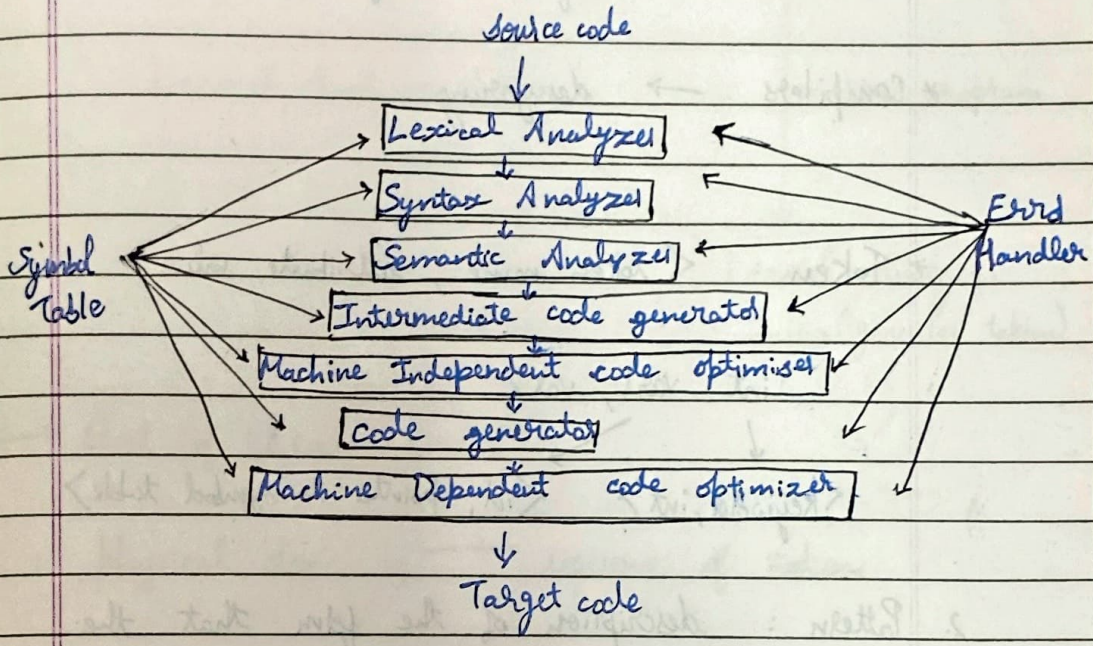
→ Library — Static
Shared

→ GCC Compilation Process



→ Symbol Table : dictionary (data structure) to store variable names, procedures with fields for the attributes of the name

→ 7 phases of compiler



→ Grouping of phases into passes

1. Single Pass Compiler
2. Two Pass Compiler
3. Three Pass Compiler

8th January, 2025

→ Syntactic sugar : syntax is designed to make things easier to read / express.

a[i] is syntactic sugar for *(a+i)

* Compilers → desugaring

1. → Token : <token name, attribute value>

int var1, var2

↓ ↘

<keyword, int> <id, pointer to symbol table>

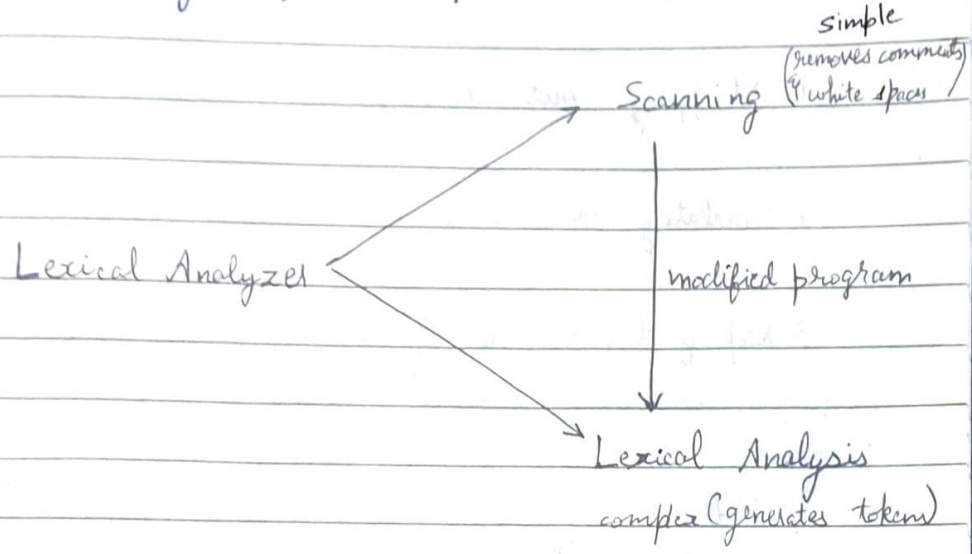
2. Pattern : description of the form that the Lexemes of a token may take.

a) keyword : seq. of characters
b) identifiers : seq. of characters with some complex structure

3. Lexeme : seq. of chars in the source program that matches the pattern for a token, i.e and instance of a token

Token	information desc. of pattern	Lexemes
keyword	characters i, f	if

Lexical Analyzer / Scanner / Lexer



→ Goal of Lexer

1. physical dev. → sequence of tokens
2. each token → logical piece of the source file (keyword, name of var)
3. each token is associated with a lexeme (actual text from source code)
4. each token → optional attributes
5. token sequence → used by parser to recover program structure.

→ Role of a lexer (1)

1. Groups the IP chars into lexemes.
2. Produces seq. of ~~char~~ tokens for each lexeme as OP
3. Sends stream of tokens to the parser for syntax analysis
4. Lexeme - identifier entered into symbol table

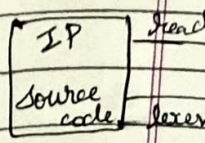
→ Role of a Lexer (2)

1. Stripping out comments & white spaces (\n, \t etc)
2. Correlating error messages by compiler → source program file
3. Keeping track: no. of \n char & associated each error message with its line no
4. Make a copy of source program with error messages inserted at appropriate positions.
5. Expansion of macros.

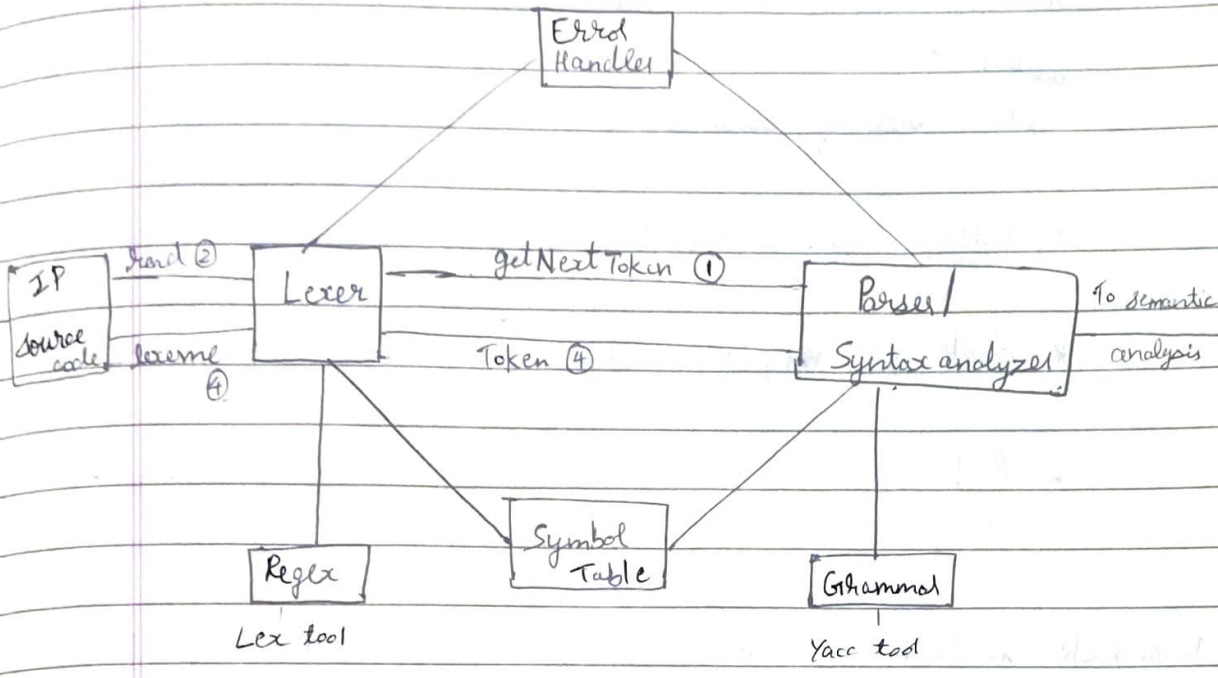
→ Reasons to seperate Lexical Analysis and Parser

1. Simplicity of design
2. Compiler efficiency ↑
3. Compiles portability ↑

Lexical Analyzer	Parser
1. scans IP program	Perfoms syntax analysis
2. identify tokens	creates an abstract rep. of code
3. inserts symbol to symbol table	update symbol table
4. generates lexical errors	generates parse tree



→ Interaction b/w Lexical Analyzer and Parser.



→ Challenge : Speed of Lexical Analyzer

* Lexer reads char to char ⇒ slow

* scenarios when lexer needs to look ahead at least one char. before a match can be announced.

Solution : The Lexical Analyzer reads from IP Buffers

- Buffer Pairs
- Sentinels

→ Input Buffers - Buffer Pairs

* 2 buffers of same size (N)
* N = size of disk blocks = 4096 bytes (usually)
* buffers are alternately reloaded
* check for longest matching
* 2 Pointers — lexeme_begin
  — forward

Drawbacks: 2 checks need to be performed:
1. For End of Buffer
2. To determine which char is read.

→ Buffer Pairs with Sentinels
Eof are added to end of each IP buffer

When lexer cannot proceed because of:
1. Spelling Error
2. Unmatched string
3. Appearance of illegal chars
4. Exceeding length of identifiers

→ Error Message Properties

1. Must Pinpoint the error correctly
2. Must be understandable
3. Must be specific
4. Must not have any redundancy

Lexical Errors

1. Spelling Error
2. Unmatched String
3. Appearance of illegal characters
4. Exceeding length of identifiers

C Language Grammar

P	:	Program Beginning
S	:	Statement
Decl	:	Declaration
Assign	:	Assignment
Cond	:	Condition
Unary Exp	:	Unary Expression
Type	:	Data type
ListVar	:	List of variables
X	:	(can take any identifier / assignment)
RelOp	:	Relational Operator

eg: language : int a, b;
a = 5;

P → S
S → Decl ; S | assign ; S | λ
Decl → Type ListVar
Type → int | float
ListVar → X | ListVar , X
X → id | Assign
Assign → id = E
E →

Arithmetic expression

$E \rightarrow E+T \mid E-T \mid T$ ④

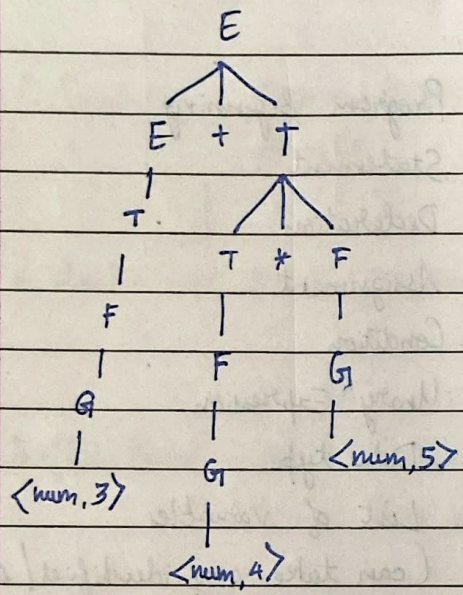
$T \rightarrow T*F \mid T/F \mid F$ ③

$F \rightarrow G \wedge F \mid G$ ② ($\wedge$ is right associative)

$G \rightarrow id \mid num \mid (E)$ ①

Prescedence & Associativity

$3 + 4 * 5$



q1. while (number > 0)
{
    factorial = factorial * number;
    -- number;
}

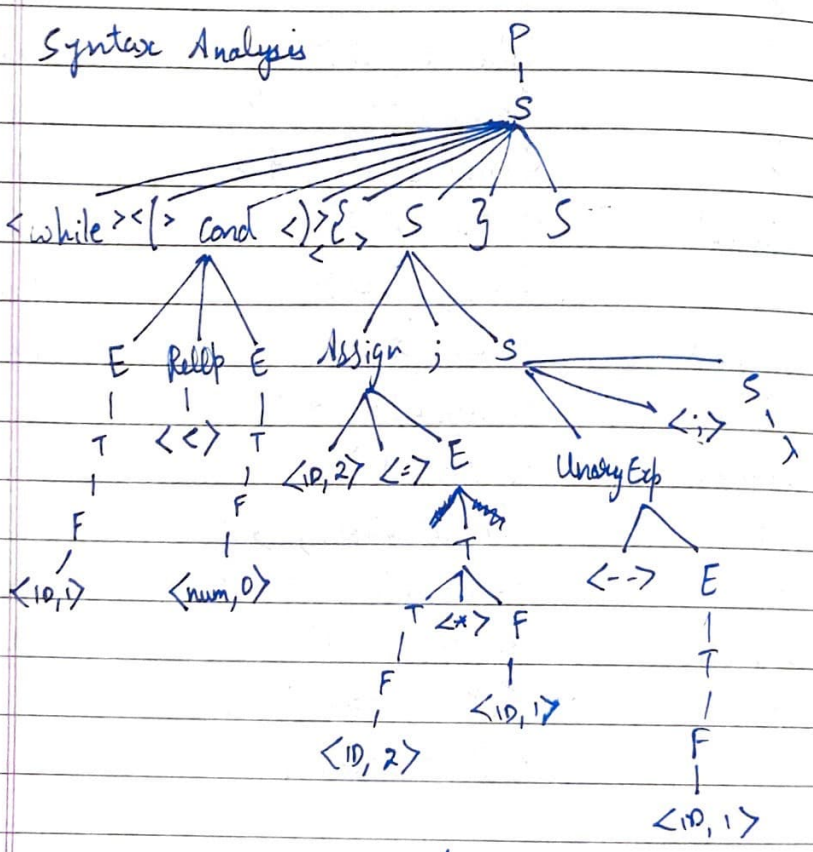
Phase 1 Lexical Analysis

Lexeme	token	Pattern
while	<keyword, while>	keyword
(	<punctuation, (>	Punctuation ~~RelOp~~
.	.	.
.	.	.
.	.	.
number	< ID, 1>	identifier
>	<RelOp, >>	RelOp
0	<Number, 0>	Number
)	<punctuation,)>	Punctuation
{	<punctuation, {>	Punctuation
factorial	< ID, 2>	identifier
=	<Assign, =>	Assign
factorial	<ID, 2>	identifier
*	<ArithOp, *>	ArithOp
number	< ID, 1 >	identifier
;	<Punctuation, ;>	Punctuation
--	<DecOp, -->	DecOp
number	< ID, 1>	identifier
;	<Punctuation, ;>	Punctuation
}	<Punctuation, }>	Punctuation

symbol table

ID	name
1	number
2	factorial

Phase 2 Syntax Analysis



Parse Tree / concrete syntax tree

eg 2.

```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
    lhs = lhs - 2 * rhs;
}
```

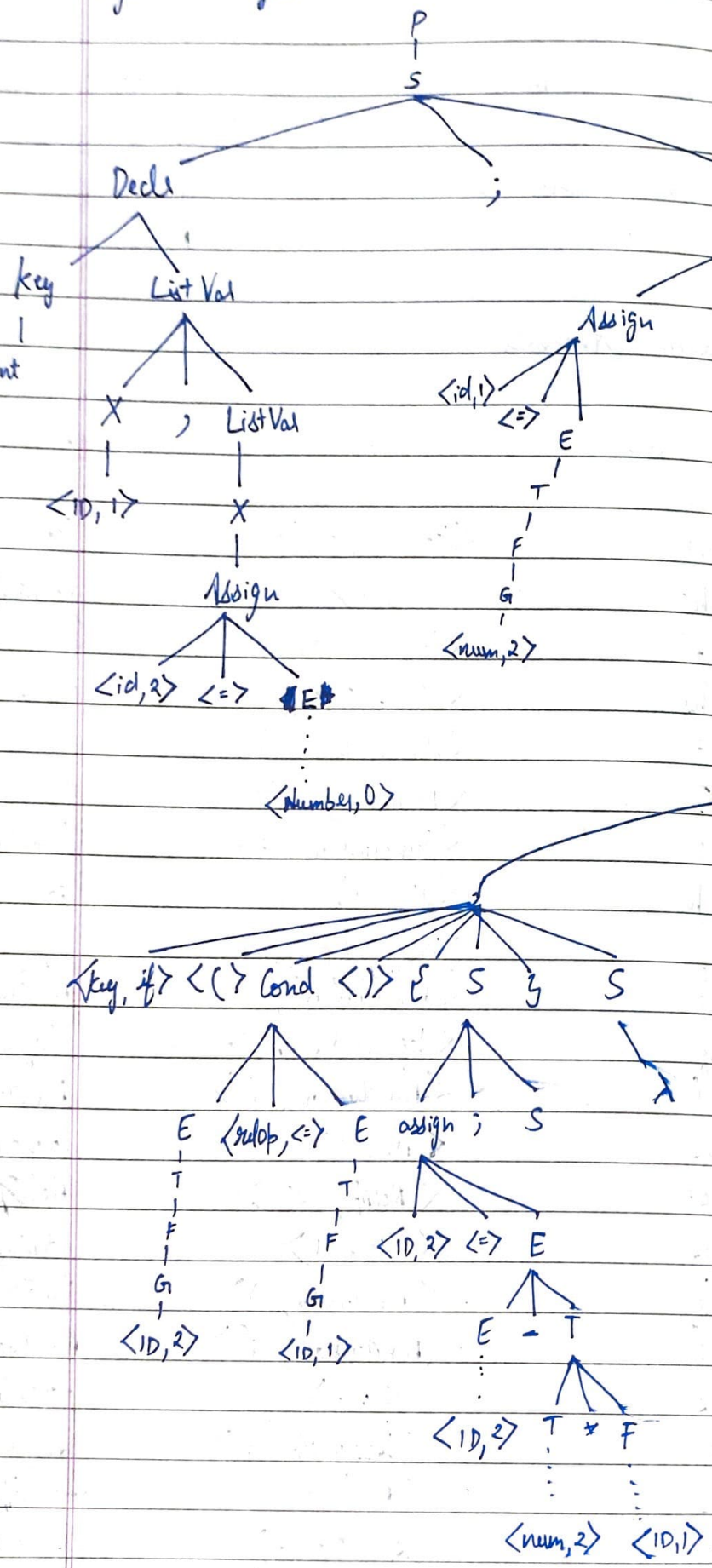
symbol table

ID	name
1	rhs
2	lhs

Phase 1 Lexical Analysis

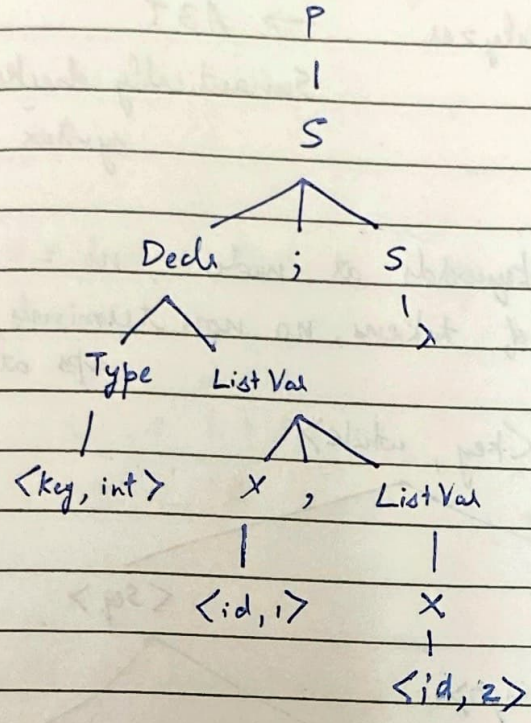
Lexeme	token	Pattern
int	<keyword, int>	keyword
rhs	<ID, 1>	identifier
,	<Punct., ,>	Punctuation
lhs	<ID, 2>	identifier
=	<AssignOp, =>	AssignOp
0	<Number, 0>	Number
;	<Punctuation, ;>	Punctuation
rhs	<ID, 1>	identifier
=	<AssignOp, =>	AssignOp
2	<Number, 2>	Number
;	<Punctuation, ;>	Punctuation
if	<keyword, if>	keyword
(	<Punctuation, (>	Punctuation
lhs	<ID, 2>	identifier
<=	<RelOp, <=>	RelOp
rhs	<ID, 1>	identifier
)	<Punctuation,)>	Punctuation
{	<Punctuation, {>	Punctuation
lhs	<ID, 2>	identifier
=	<AssignOp, =>	AssignOp
lhs	<ID, 2>	identifier
-	<ArithOp, ->	ArithOp
2	<Number, 2>	Number
*	<ArithOp, *>	ArithOp
rhs	<ID, 1>	identifier
;	<Punc, ;>	Punctuation

Phase 2: Syntax Analysis



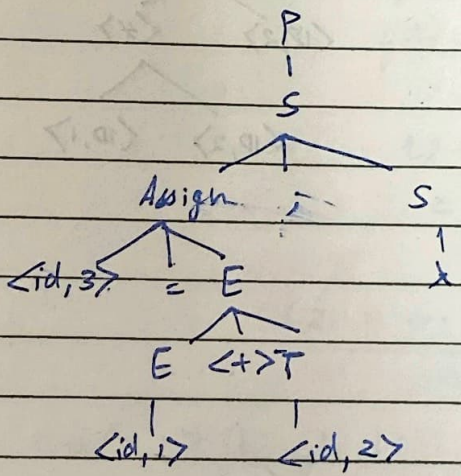
13th January, 2025

eg: int a, b;

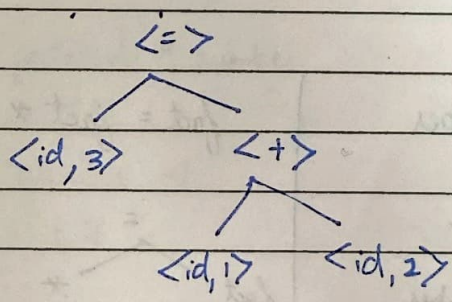


eg: c = a + b;
3 1 2

syntax tree:



AST

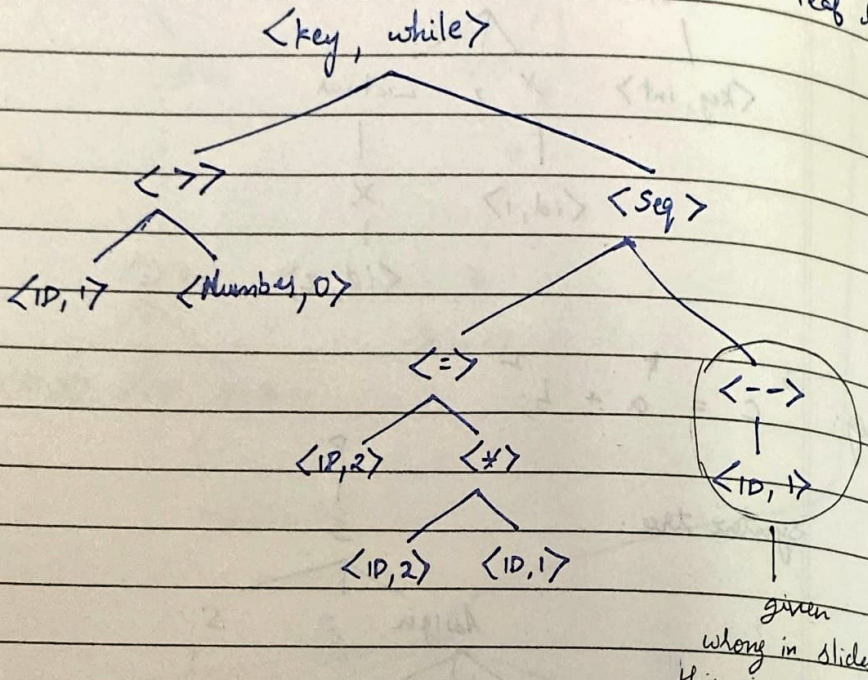


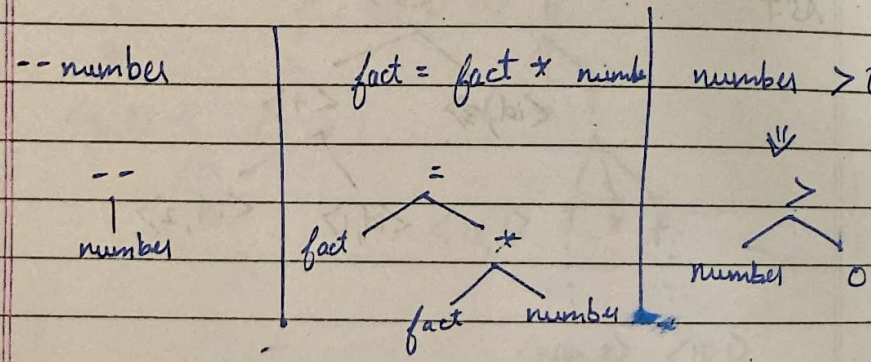
q1.

Phase 3 Semantic Analyzer → AST

Semantically checked syntax tree

* In AST no keywords at nodes, no =
  └ only set of tokens, no non terminals, punctuations ops at leaf level





Phase 4- Intermediate code generator

op: 3AC

_ = _ □ _

eg: a = b + c * d

t1 = c * d
t2 = b + t1
a = t2

eg: if (a > b) {
c = a + b;
}

* In 3AC only if construct is allowed no while, for

t1 = a > b
if t1 goto L1
goto L2

L1: t2 = a + b
c = t2

L2: .....

L0: if number > 0 goto L1
goto L2

L1: t1 = factorial * number
factorial = t1
t2 = number - 1
number = t2
goto L0

L2: ...

Phase 5 Machine Independent Code Optimization

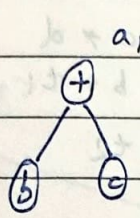
OP: DAG Optimization

used to eliminate common sub expression

eg: $a = b + c$

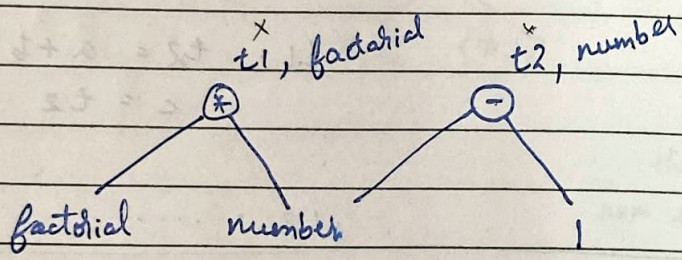
$d = b + c$

$e = d$



basic block: set of seq. of statements without branching

construct DAG for basic blocks




```
L0 : if number > 0 goto L1
     goto L2
L1 : factorial = factorial * number
     number = number - 1
     goto L0
L2 : ....
```

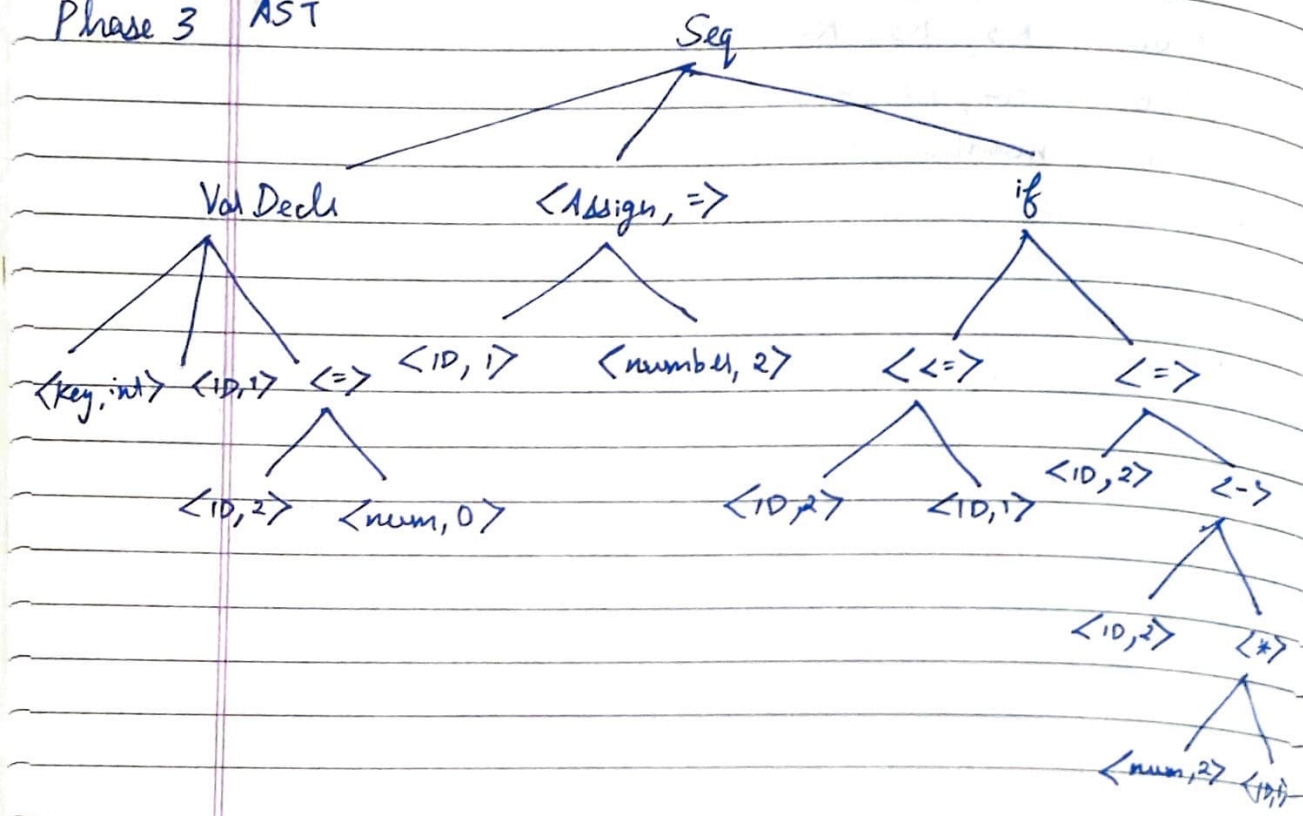
Phase 6 Code Generation

OP : Assembly code

```
    LD   R1, #0
L0: LD   R4, number
    SUB  R2, R4, R1
    BLTZ R2, L2
    LD   R3, factorial
    MUL  R3, R3, R4
    SUB  R4, R4, #1
    ST   number, R4
    goto L0
L2: ...
```

eg2:
```
int rhs, lhs = 0;
rhs = 2;
if (lhs <= rhs)
{
    lhs = lhs - 2*rhs;
}
```

Phase 3 AST



Phase 4 Intermediate code generation

OP 3AC

~~t1 = rhs~~
lhs = 0
rhs = 2
t0 = lhs <= rhs
if t0 goto L1
goto L2

L1:

L2:

Phase 5

lhs

Phase 6 Cod

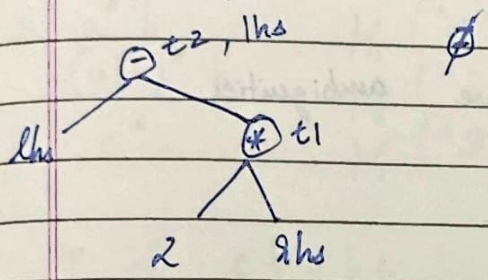
MOV
~~ADD~~ S
MOV
ST
SUB
BLTE
Qb L
L1: LD
MUL
SUB
ST

L2:

L1: t1 = 2 * rhs
t2 = lhs - t1
lhs = t2

L2: ...

Phase 5 DAG optimization



lhs = 0
rhs = 2
if (t0 = lhs <= rhs)
if t0 goto L1
goto L2
L1 : t1 = 2 * rhs
lhs = lhs - t1
L2 : next

Phase 6 Code Generation

```
MOV  R1, #0
ST   lhs, R1
MOV  R2, #2
ST   rhs, R2
SUB  R3, R1, R2
BLTEZ R3, L1
goto L2
L1: LD   R4, #2
MUL  R4, R2
SUB  R5, R1, R4
ST   R5, lhs
L2: ...
```

q3.
```
n = 23
for (i = 0 ; i < n; i++)
  {
     sum = sum * i;
  }
```

How does Lexer resolve ambiguities

* Maximal Munch rule
  - longest match possible
  - if more than 1 match is found then earliest match is considered.

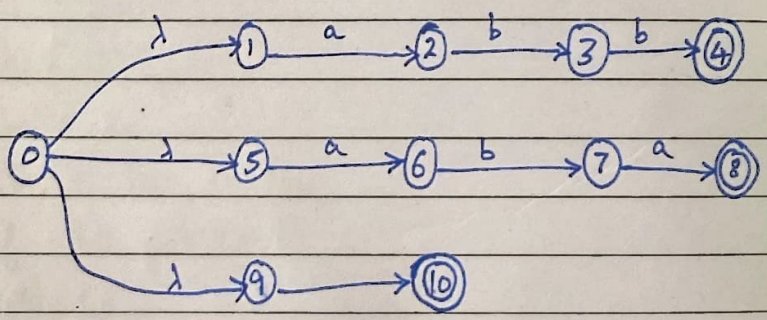
eg: 1.
```
abb     if ("1")
aba     if ("2")
a       if ("3")
```

i) IP a , OP: 3

ii) IP ababa, OP: a2 b3

15th January, 2025

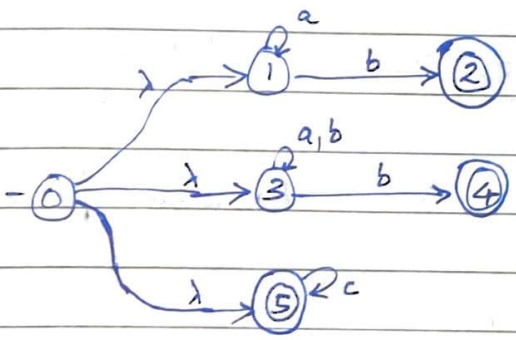
NFA



	a	b	b
0	1		
1	2	3	4*
5	6	7	–
9	10*	–	

2. $a^* b$ — bf ("1")
$(a|b)^* b$ — bf ("2")
c^* — bf ("3")

IP: i) c b a b c = 3 2 3
ii) c b b b b a c = 3 2 a 3
iii) ~~c^*~~ =



	c ↓	b ↓	b	b	b	a	c
0	–	–				0	0
1	–	2				1	1
3	–	3* 4*	3* 4*	3 4*	3 4*	3	3
5	5*	–				5	5*

OP = 132 IP = $a^* b\ c^*\ (a|b)^* b$ ✗

↓

a b c b b ✓

~~b* c b~~ a a b c b b b ✓

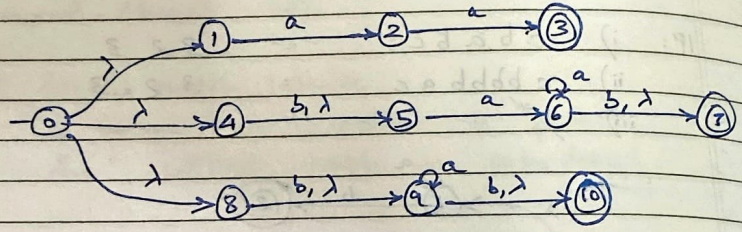
1 3 2

q3 a a bf ("1")
b? a +b? bf ("2")
b? a* b? bf ("3")

i) bbbaabb
ii) string for 123
iii) string for 321

* - 0 or more
\+ - 1 or more
? - 0 or 1

i) 323



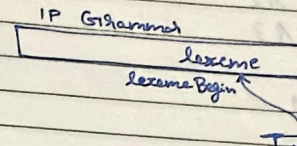
ii) aa ba aaaaaa X No string possible
1 2

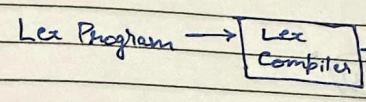
iii) baaaaab aab aa
3 2 1

q4. a* bf ("1")
ab bf ("2")
bb bf ("3")

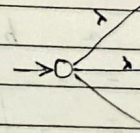
a a b b b
1 3 b

Structure of Lexical Analyzer



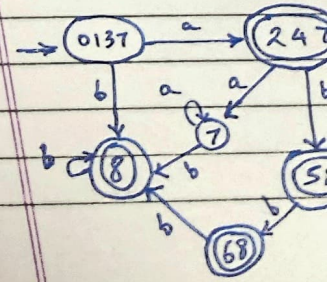


q1. a A1
abb A2
a* b+ A3

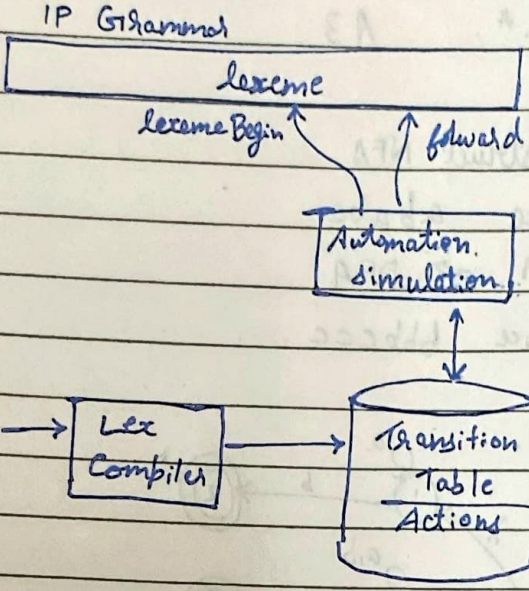


Step 1 Construct NFA :

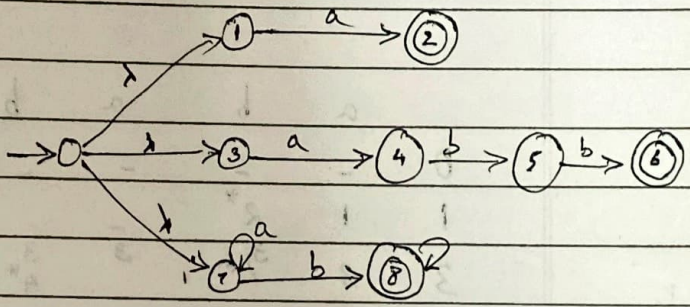
Step 2 NFA → DFA



Structure of Lexical Analyzer Generator



q1. a A1
abb A2
a* b+ A3

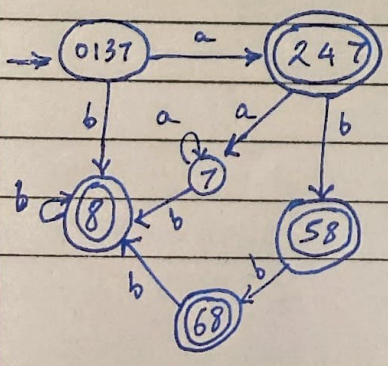


Step 1 Construct NFA :

Step 2 NFA → DFA

q:	a	a	b	a
0	-			
1	2*	-		
3	4	-		
7	7	7	8*	

0	-
1	2*
3	4
7	7



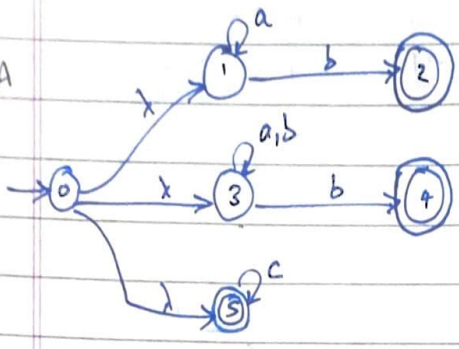
q2. a^*b A1

$(a|b)^*b$ A2

c^* A3

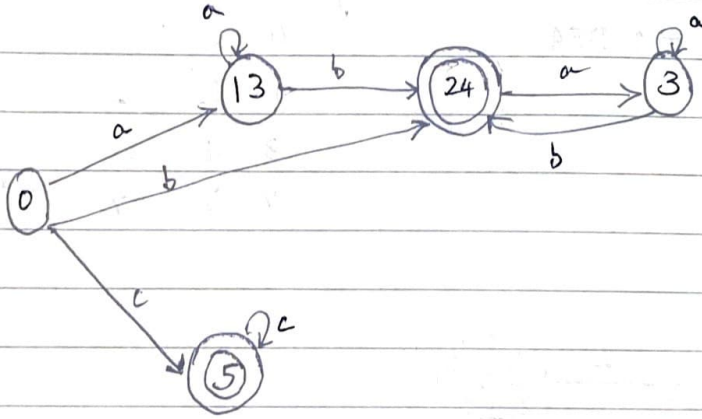
1. Construct NFA
2. Trace ababc
3. NFA → DFA
4. Trace bbbccc

NFA



	a	b	a	b		c
0	-	-	-	-	0	-
1	1	2^*	-	-	1	-
3	3	$3\ 4^*$	3	$3\ 4^*$	3	-
5	-	-	-	-	5	5^*

DFA



20th January, 2025

Lex Specification file

definitions (not mandatory)

start of rules → %%

regular exp & associated actions (rule)

%%

user routines

compile lex program → lex.yy.c

|

handles only patterns

eg:

```
%%
. ECHO;
\n ECHO;
%%
```

definition

```
%{
%}
```

eg:

```
%{
    int yylineno = 1;
%}
%%
(.*)\n  printf("%4d\t%s", yylineno, yytext);
%%
```

regular definition / Regular expression definition - giving meaningful name to defintion ↑ readability

digit [0-9]

letter [_A-Za-z]

[^ ...] → match except the following chars

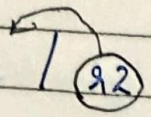
^ beginning of line

abc : matches "abc" "abcabc"

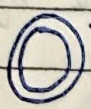
[abc] : matches any char a or b or c

→ Lookahead operator in Lex (/)

* certain pattern is req. to be matched only when it is followed by certain other char



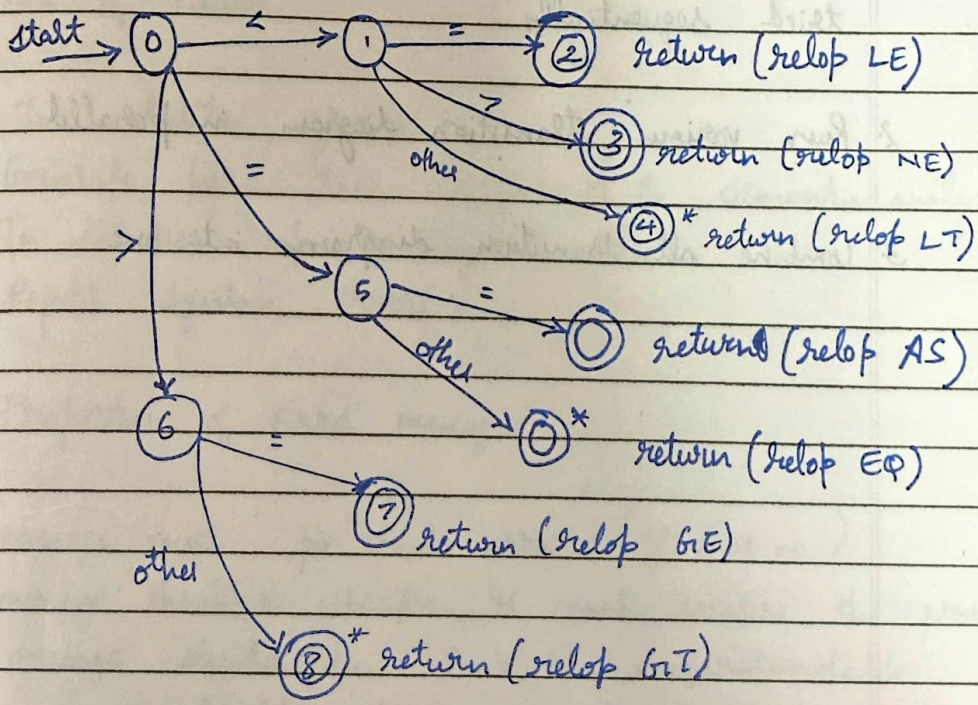
r1 is valid token only if r2 occurs

→  * - retract

go back & return

→ Transition diagram for relational operators

relop | < | <= | > | >= | <> | ==



* white spaces are not considered as tokens by parser hence while matching for whitespace don't return

21st January, 2025

→ Keywords vs Identifiers

Ways to handle keywords that look like identifiers

1. Install the reserved words in the symbol table before hand

2. Create seperate transition diagram

→ How does lexer implement all transition diagrams

1. Arrange transition diagrams for each token to be tried sequentially

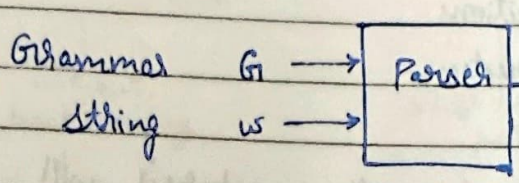
2. Run various transition diagram in parallel

3. Combine all transition diagrams into one

January, 2025

Parser - Syntax Analyzer

int ++ a, b; → acceptable by lexer
does not check grammar



IP read from left to write

Lexical Analyzer → Parser

→ Role of Parser

* To validate syntax
* Generate parse tree & pass it to semantic analyzer
* To store info in symbol tree
* Report syntax errors.

→ Properties of Error messages

* message must pin point error (eg: line no.)
* message should be specific & must localise the problem
* message should be clear & understandable
* message should not be redundant

→ Error Recovery in Parsers

1. Panic Mode Recovery
2. Phrase Level Recovery
3. Error Productions
4. Global Correction

- Panic Mode recovery : parser will discard IP symbols (skip) until delimiter is ; or }
then restart parsing process.

such delimiters are called : synchronizing tokens

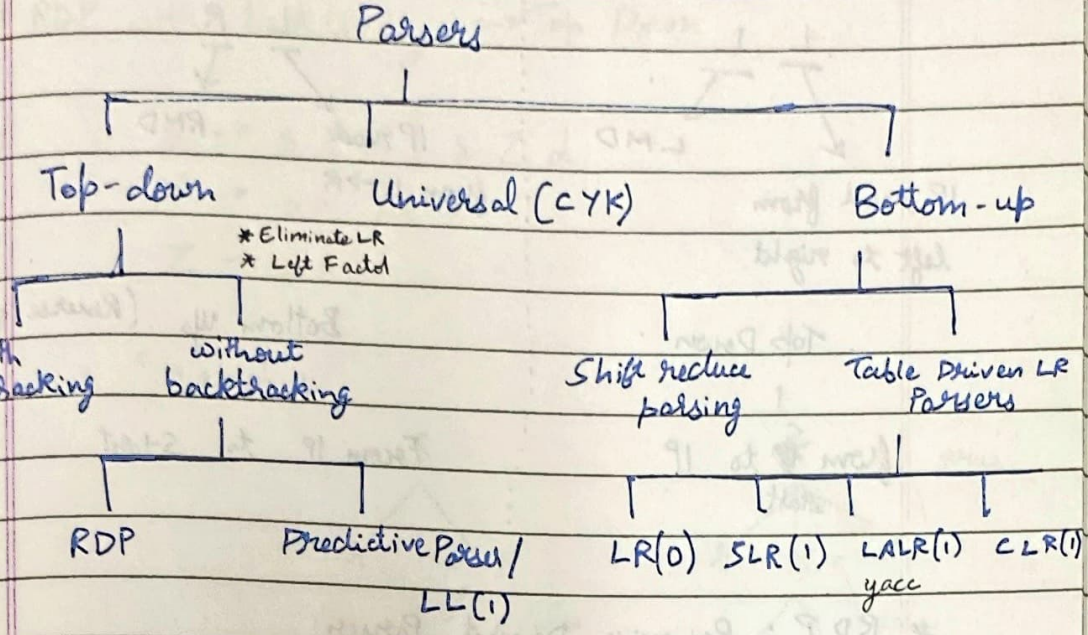
- Phrase Level recovery : performing local corrections in IP program in case it contains errors.

- Error Productions : specify grammar for known common mistakes

- Global Correction : * extremely hard to implement
* use algorithm to correct errors

Disadvantages : * closest correct program may not be exactly what programmer intended to write
* slows down parsing of correct programs.

2nd January, 2025



Top Down

LMD - Left Most Derivation

for expression

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to id \mid num$

$E \xrightarrow{LMD} \underline{E} + T$

$E \to \underline{T} + T$

$T \to \underline{F} + T$

$F \to id + \underline{T}$

$T \to id + T * \underline{F}$

$T \to id + \underline{F} * F$

$F \to id + id * \underline{F}$

$F \to id + id * id$

RMD - Bottom up Parser is reverse of RMD

$E \xrightarrow{RMD} E + \underline{T}$

$T \to E + T * \underline{F}$

$F \to E + \underline{T} * id$

$T \to E + F * id$

$F \to \underline{E} + id * id$

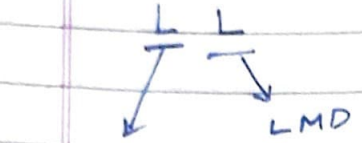
$E \to \underline{T} + id * id$

$T \to \underline{F} + id * id$

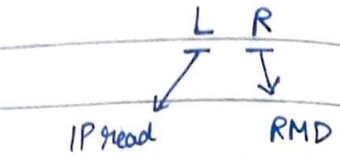
$F \to id + id * id$

Bottom up Parser ↑ reverse of RMD

You are awesome



IP read from left to right



from L → R

Top Down

|

from S to IP

Start

Bottom Up (Reverse of RMD)

|

From IP to Start

* RDP : Recursive Descent Parser

|

we have to write procedures for non terminals

LL (1)

no. of lookahead symbols to take decision

Issues : * RDP cannot work for Left recursive grammar

|

E → E + T

```
E() {
    if E():
```

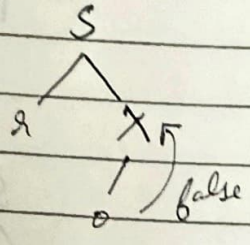
× infinite calls

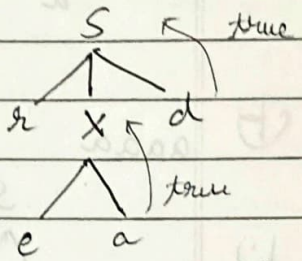
* Too much backtracking

→ RDP with backtracing ↔ Top Down

eg:
$S \rightarrow \text{r}\,X\,\text{d} \mid \text{r}\,Z\,\text{d}$
$X \rightarrow \text{oa} \mid \text{e a}$
$Z \rightarrow \text{ai}$

w = "read"

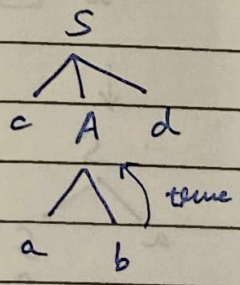


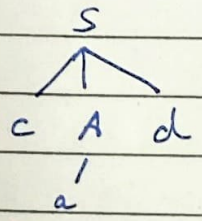


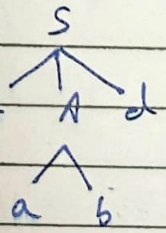
$S \rightarrow cAd$
$A \rightarrow ab \mid a$

$A \rightarrow a \mid ab$

IP w = "cabd"

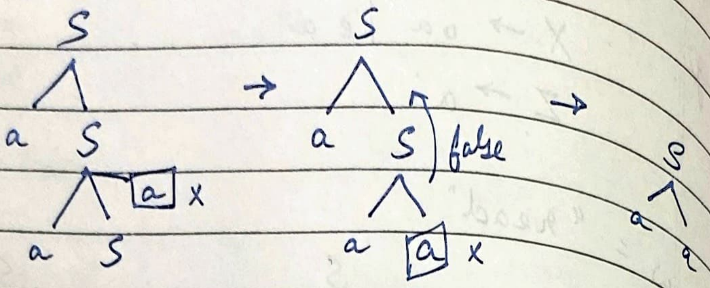




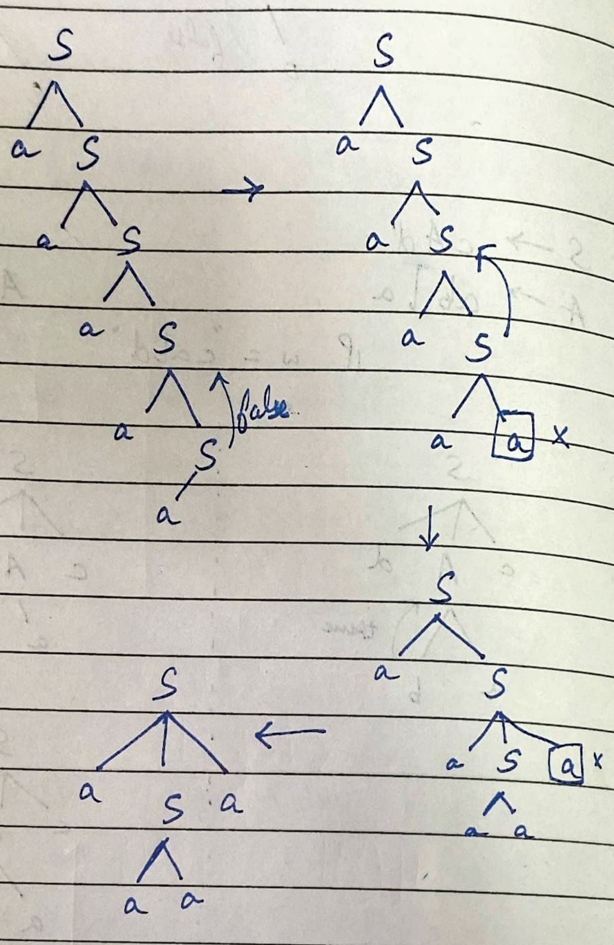


Q: $S \rightarrow aSa \mid aa$ $L(G) = \{a^{2n}, n \geq 1\}$

aa



aaaa



U ARE THE GOD THE G

23rd January, 2025

Top-Down Parser without backtracking

2 steps : 1. Left Factoring
2. Elimination of Left Recursion

→ Left Factding : Remove common prefix to avoid same symbol being scanned again

→ Elimination of Left Recursion

eg1. $A \to Aa \mid b$ — Left recursive

$A \to bA'$

$A' \to aA' \mid \lambda$ — Not Left recursive

eg 2. $S \to Sa \mid d \mid e$

$S \to dS' \mid eS'$

$S' \to aS' \mid \lambda$

eg 3. $S \to Aab$, $A \to Sd \mid \lambda$ — indirect Left Recursion

$A \to Aabd \mid X \mid b$

$A \to$ ~~$Ad \mid bR$~~ $abdA' \mid \lambda$

$A' \to$

$A \to Aabd$

$A \to bA'$

$A' \to abdA' \mid \lambda$

eg 4. $E \rightarrow E + T \mid E$

$\Downarrow$

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \lambda$

q1. $A \rightarrow Bxy \mid x$

$B \rightarrow CD$

$C \rightarrow A \mid c$

$D \rightarrow d$

Eliminate Left Recursion

$A \rightarrow Bxy \mid x$

$B \rightarrow CD$

$C \not\rightarrow Bxy \mid x \mid c$

$D \rightarrow d$

$\Rightarrow$

$A \rightarrow Bxy \mid x$

$B \rightarrow CD$

$C \rightarrow C Dxy \mid x \mid c$

$D \rightarrow d$

$\Downarrow$

$A \rightarrow Bxy \mid x$

$B \rightarrow CD$

$C \rightarrow xC' \mid cC'$

$C' \rightarrow DxyC' \mid \lambda$

$D \rightarrow d$

q2. $S \rightarrow aAcB$

$A \rightarrow Ab \mid b \mid bc$

$B \rightarrow d$

$\Downarrow$

$S \rightarrow aAcB$

$A \rightarrow bA' \mid bcA'$

$A' \rightarrow bA' \mid \lambda$

$B \rightarrow d$

q3. $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

→ Computing First Set

q4. $S \rightarrow Sa \mid Sb \mid c \mid d$

$S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

$S \rightarrow cS' \mid dS'$

$S' \rightarrow aS' \mid bS' \mid \lambda$

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

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

→ First Set

↓

used to choose the production without checking all cases.

$S \rightarrow ab \mid cd$

if string starts with c we need not check for first ab.

* In the first set we can include λ
* In the follow set never include λ

Find first set of all non-terminals

First Set

i) $S \rightarrow a \mid b$ — First(S) = {a, b}

ii) $S \rightarrow Ea \mid \lambda$ — First(S) = {λ, c, d}
$E \rightarrow c \mid d$ — First(E) = {c, d}

iii) $S \rightarrow ET$ — First(S) = {a, b, c, λ} (since E, T both have λ)
$E \rightarrow a \mid b \mid \lambda$ — First(E) = {a, b, λ}
$T \rightarrow c \mid \lambda$ — First(T) = {c, λ}

iv) $E \rightarrow TE'$ — First(E) = ~~{id, num, +, *, λ}~~
$E' \rightarrow +TE' \mid \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \lambda$
$F \rightarrow id \mid num \mid (E)$

First(E') = {

First Set	E	E'	T	T'	F
	id	+	id	*	id
	num	λ	num	λ	num
	(		(		(

v) $S \rightarrow aBDh$

$B \rightarrow aC$

$C \rightarrow bC \mid \lambda$

$D \rightarrow EF$

$E \rightarrow g \mid \lambda$

$F \rightarrow f \mid \lambda$

First (S) = { a }

First (B) = { c }

First (C) = { b, λ }

First (D) = { g, f, λ }

First (E) = { g, λ }

First (F) = { f, λ }

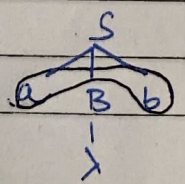
→ Follow Set - when we can nullify

to decide where we can apply λ to terminate

eg: $S \rightarrow aBb$

$B \rightarrow c \mid \lambda$

ab



$S \rightarrow E +$

$S \rightarrow$

* Always start follow set with $

case i) $S \rightarrow E +$ follow(E) = +
case ii) $S \rightarrow E T$ follow(E) = first(T)
case iii) $S \rightarrow E T$ follow(T) = follow(S)

Q
$E \rightarrow TE'$
$E' \rightarrow +TE' | \lambda$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \lambda$
$F \rightarrow id | num | (E)$

Follow Set

E	E'	T	T'	F
$	$	+	+	*
)	)	$	$	+
		)	)	$
				)

q. $S \to ACB \mid Cbb \mid Ba$
$A \to da \mid BC$
$B \to g \mid \lambda$
$C \to h \mid \lambda$

First (S) = {d, g, h, a, b}
First (A) = {d, g, h, λ}
First (B) = {g, λ}
First (C) = {h, λ}

Follow (S) = {$}
Follow (A) = {h, $, a}
Follow (B) = {h, $, a, g}
Follow (C) = {b, g, $, h}

27th January, 2025

* LL = LL(1) by default 1 look ahead

Constructing LL(1) Parsing Table

q1. $E \to TE'$
$E' \to +TE' \mid \lambda$
$T \to FT'$
$T' \to *FT' \mid \lambda$
$F \to num \mid id \mid (E)$

First E E' T T' F Follow: E E' T T' F

* always load stack with $
* always put $ at end of IP buffer

IP is valid ⇒ only $ in stack & IP buffer

* fill all production rules seperately in table

$E \to TE'$ fill this in all places of first (E)

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

$E' \to \lambda$ take follow E'

* since no multiple entries ⇒ grammar is LL(1)

left factored, no left recursion

Stack	IP Buffer	Action
$E\$$	$id+id\$$	$M[E, id] \to$ pop E & push TE'
$TE'\$$	$id+id\$$	$M[T, id] \to$ pop T & push FT'
$FT'E'\$$	$id+id\$$	$M[F, id] \to$ pop F & push id
$id\ T'E'\$$	$id+id\$$	$M[id, id] \to$ pop id both places
$T'E'\$$	$+id\$$	$M[T', +] \to$ pop T' & push λ
$E'\$$	$+id\$$	$M[E', +] \to$ pop E' & push $+TE'$

~~+~~TE' $	~~+~~id $	M[+,+] → pop + both places
TE' $	id $	M[T, id] → pop T push FT'
FT'E' $	id $	M[F, id] → pop F push id
~~id~~ T'E' $	~~id~~ $	M[id, id] → pop id both places
T'E' $	$	M[T', $] → pop T' push λ
E' $	$	M[E', $] → pop E' push λ
$	$	accept

q2. $S \to a \mid (L)$ Construct LL(1) table

$L \to L, S \mid S$ — this might

this will not cause ∞ loop as not starting

First ~~S~~ ~~L~~

$S \to a \mid (L)$

$L \to L, S \mid a \mid (L)$

⇓ remove Left recursion

$S \to a \mid (L)$

$L \to aL' \mid (L)L'$ } free from LR

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

First:

S	L	L'
a	a	,
(	(	λ

Follow:

S	L	L'
$	)	)
,		

→ follow(L'):)

a ( ) , $

S	S→a	S→(L)			
L	L→aL'	L→(L)L'			
L'			L'→λ	L'→,SL'	

q3. $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \lambda$

$B \rightarrow \lambda$

Is this grammar in LL(1)

Also compute first & follow sets

First:

S	A	B
a	λ	λ
b		
~~ab~~		

Follow

S	A	B
$	a	a
	b	b

	a	b	$
S	$S \rightarrow AaAb$	$S \rightarrow BbBa$	
A	$A \rightarrow \lambda$	$A \rightarrow \lambda$	
B	$B \rightarrow \lambda$	$B \rightarrow \lambda$	

No multiple entries

No left recursion,

No left factoring $\Rightarrow$ LL(1)

Parsing using an LL(1) Parsing Table

Initially,

Stack contain start symbol $

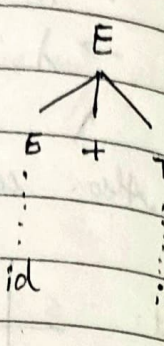
IP Buffer contain IP string w

Stack	IP Buffer	Action
S$	w$	

Finally,

Stack	IP Buffer	Action
$	w	accept

IP : id + id



Stack	IP Buffer	Action
E $	id + id $	

---

eg 2. $S \to a \mid (L)$
$L \to aL' \mid (L)L'$
$L' \to ,SL' \mid \lambda$

	a	(	)	,	$
S	$S \to a$	$S \to (L)$			
L	$L \to aL'$	$L \to (L)$			
L'			$L' \to \lambda$	$L' \to ,SL'$	

IP: (a, a)

Stack	IP Buffer	Action
S $	(a,a) $	M[S, (] → pop S, push (L)
(L) $	(a,a) $	M[(, (] → pop (both
L) $	a,a) $	M[L, a] → pop L push aL'
aL') $	a,a) $	M[a, a] → pop a both
L') $	,a) $	M[L', ,] → pop L' push SL'
SL') $	,a) $	M[S, ,] → pop S ~~push~~ both

S L') $	a) $	M[S, a] → pop S, push a
a L') $	a) $	M[a, a] → pop a both
L') $	) $	M[L',)] → pop L' push λ
) $	) $	M[ ),)] → pop) both
$	$	accept

28th January, 2025

eg: S → iCtS | iCtSeS | a
C → b

A B

	i	a	e	t	$
S	S → iCtS S → iCtSeS				
C					

Not LL(1)

* First (A) ∩ First (B) = ϕ for no multiple entries

eg: S → a | λ
B → bSa

	a	b	$
S	S → a S → λ		
B			

* First (A) ∩ Follow (S) = ϕ for no multiple entries

First

S	A	B
a	λ	λ
b		

Follows

S	A	B
$	a	a
	b	b

eg3. $S \to AaAb \mid BbBa$

$A \to \lambda$

$B \to \lambda$

~~First (AaAb)~~ = First(A) = {a}

First (BbBa) = First(B) = {λ} $\Rightarrow \cap$

eg4. $S \to iCtS \mid iCtSeS \mid a$

$C \to b$

First (iCtS) = {i}

First (iCtSeS) = {i} $\Rightarrow$ Not LL(1)

modified grammar:

$S \to iCtSS' \mid a$

$S' \to \lambda \mid eS$

$C \to b$

foll (S) = {$, e}

foll (S') = {$, e}

= foll (S)

when λ: i) $i \cap a = \phi$

ii) first(S') $\cap$ follow(S') = {e}

$\Rightarrow$ Not LL(1)

	a	i	t	e	$
S'				S'→eS S'→λ	S'→λ

Follow

eg 5: $S \to AB$
$A \to a \mid \lambda$
$B \to b \mid \lambda$

First

S	A	B
a	a	b
b	λ	λ
λ		

Follow

S	A	B
$	b	$
	$	

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	~~$A \to \lambda$~~ ~~$B \to b$~~	
B		$B \to b$	$B \to \lambda$

eg 6: $S \to ABCDE$
$A \to a \mid \lambda$
$B \to b \mid \lambda$
$C \to c$
$D \to d \mid \lambda$
$E \to e \mid \lambda$

First

S	A	B	C	D	E
a	a	b	c	d	e
b	λ	λ		λ	λ
c					

Follow

S	A	B	C	D	E
$	b	c	d	e	$
	c		e	$	
			$		

→ LL parsing table

	a	b	c	d	e	$
S	S→ABCDE	S→ABCDE	S→ABCDE			
A	A→a	A→λ	A→λ			
B		B→b	B→λ			
C			C→c			
D				D→d	D→λ	D→λ
E					E→e	E→λ

→ w = cde

Stack	I/P Buffer	Action
S$	cde$	M[S, c] → pop S, push ABCDE
ABCDE$	cde$	M[A, c] → pop A
BCDE$	cde$	M[B, c] → pop B
CDE$	cde$	M[C, c] → pop C, push c
cDE$	cde$	M[c, c] → pop both.
DE$	de$	M[D, d] → pop D, push d
dE$	de$	M[d, d] → pop d
E$	e$	M[E, e] → pop E, push e
e$	e$	M[e, e] → pop e
$	$	accept

Error Recovery in LL(1) Parser

→ Panic Mode recovery

*

eg:

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

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

* Find follow of each non-terminal.
if that cell is not filled put sync
if filled leave it as is

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

First

E	E'	T	T'	F
id	+	id	*	id
num	λ	num	λ	num
(		(		(

Follow

E	E'	T	T'	F
$	$	+	+	*
)	)	$	$	+
		)	)	$
				)

Rule 1.* if "sync" and only 1 non terminal in the stack ⇒ ignore IP symbol

Rule 2. if "sync" and more than 1 non terminal in the stack ⇒ pop non terminal

IP:)id * id + id $

stack	IP	Action
E$	)id * + id $	sync, skip
E$	id * + id $	M[E, id] → pop E push TE'
TE'$	id * + id $	M[T, id] → pop T push FT'
FT'E'$	id * + id $	M[F, id] → pop F push id
id T'E'$	id * + id $	match
T'E'$	* + id $	M[T', *] → pop T push *FT'
* F T'E'$	* + id $	match
F T'E'$	+ id $	sync, pop F
T'E'$	+ id $	
E'$	+ id $	
+ T E'$	+ id $	
TE'$	id $	
FT'E'$	id $	
id T'E'$	id $	
E'$	$	
$	$	

eg2: $S \to a \mid (L)$

$L \to aL' \mid (L)L'$

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

First

S	L	L'
a	a	,
(	(	λ

Follows

S	L	L'
,	)	)
$		
)		

LL(1) Parsing Table

	a	(	)	,	$
S	$S \to a$	$S \to (L)$	sync	sync	sync
L	$L \to aL'$	$L \to (L)L'$	sync		
L'			$L' \to \lambda$	$L' \to ,SL'$	

$w = a(a)$

Stack	IP	Action
S$	a(a)$	pop S push a
~~a~~$	~~a~~(a)$	pop a
$	(a)$	

30th January, 2025

eg $E \to TE'$

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

$T \to FT'$

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

$F \to id$

$w = id(+)id$

stack	IP Buffer	Action
$E\$$	$id(+)id\$$	pop E push TE'
$TE'\$$	$id(+)id\$$	pop T push FT'
$FT'E'\$$	$id(+)id\$$	pop F push id
$id\ T'E'\$$	$id(+)id\$$	pop id
$T'E'\$$	$(+)id\$$	Syntax Error

parser cannot proceed.

eg 5. $S \to AB$

$A \to a \mid \lambda$

$B \to b \mid \lambda$

LL(1) parsing

First

S	A	B
a	a	b
b	λ	λ
λ		

Follow

S	A	B
$	b	$
	$	

LL(1) parsing table

	a	b	$
S	$S \to AB$	$S \to AB$	$S \to AB$
A	$A \to a$	$A \to \lambda$	$A \to \lambda$
B		$B \to b$	$B \to \lambda$

here we have no λ in this table but if both AB become λ we have to check follow of S

parse the strings i) λ
ii) b

i)

stack	IP Buffer	Action
S$	$	M[S,$] → pop S push AB
AB$	$	M[A,$] → pop A ~~push~~
B$	$	M[B,$] → pop B
$	$	accept

ii)

stack	IP Buffer	Action
S$	b$	M[S,b] → pop S push AB
AB$	b$	M[A,b] → pop A ~~push~~
B$	b$	M[B,b] → pop B push b
b$	b$	pop b
$	$	accept

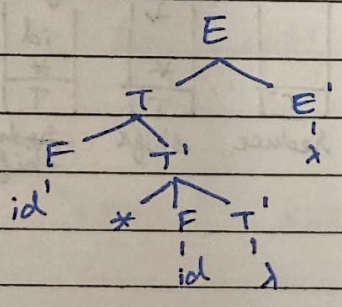
Bottom-Up Parsers

eg: E → E + T | T
T → T * F | F
F → id

of Top Down →

E → TE'
E' → +TE' | λ
T → FT'
T' → *FT' | λ
F → id

w = id * id



(d)

E → TE'
→ FT'E'
→ id T'E'
→ id * FT'E'
→ id * id T'E'
→ id * id E'
→ id * id