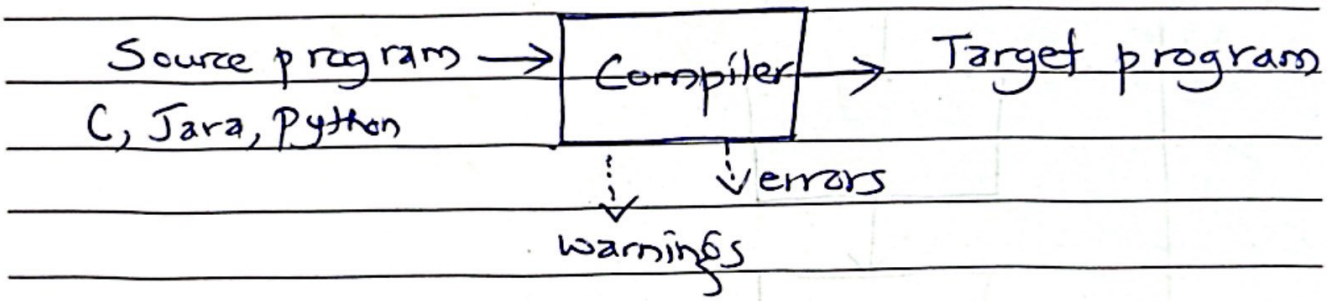
S M T W TH F S Date .....

COMPILER DESIGN

Compiler:

It is a software/program that converts a program written in High level language (source language) to a low level language (object / Target language).

- It also reports errors present in source programs.



Types of compilers.

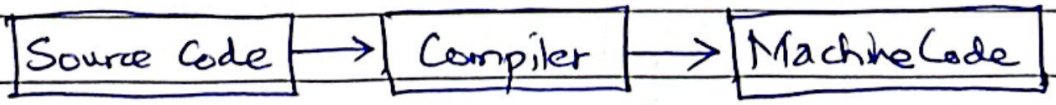
1). Single pass compilers

It is type of compiler that processes the source code only once.

E.g. Early versions of Pascal compilers

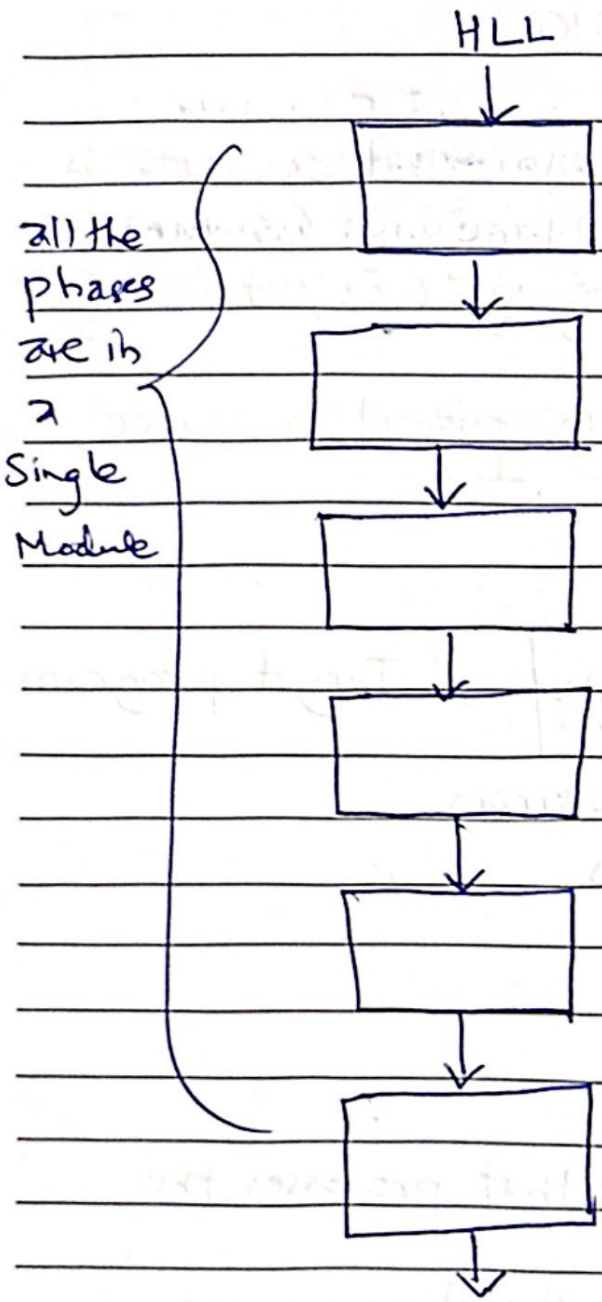
Advantage: faster compilation

Disadvantage: Limited optimization capabilities



S M T W TH F S

Date .....

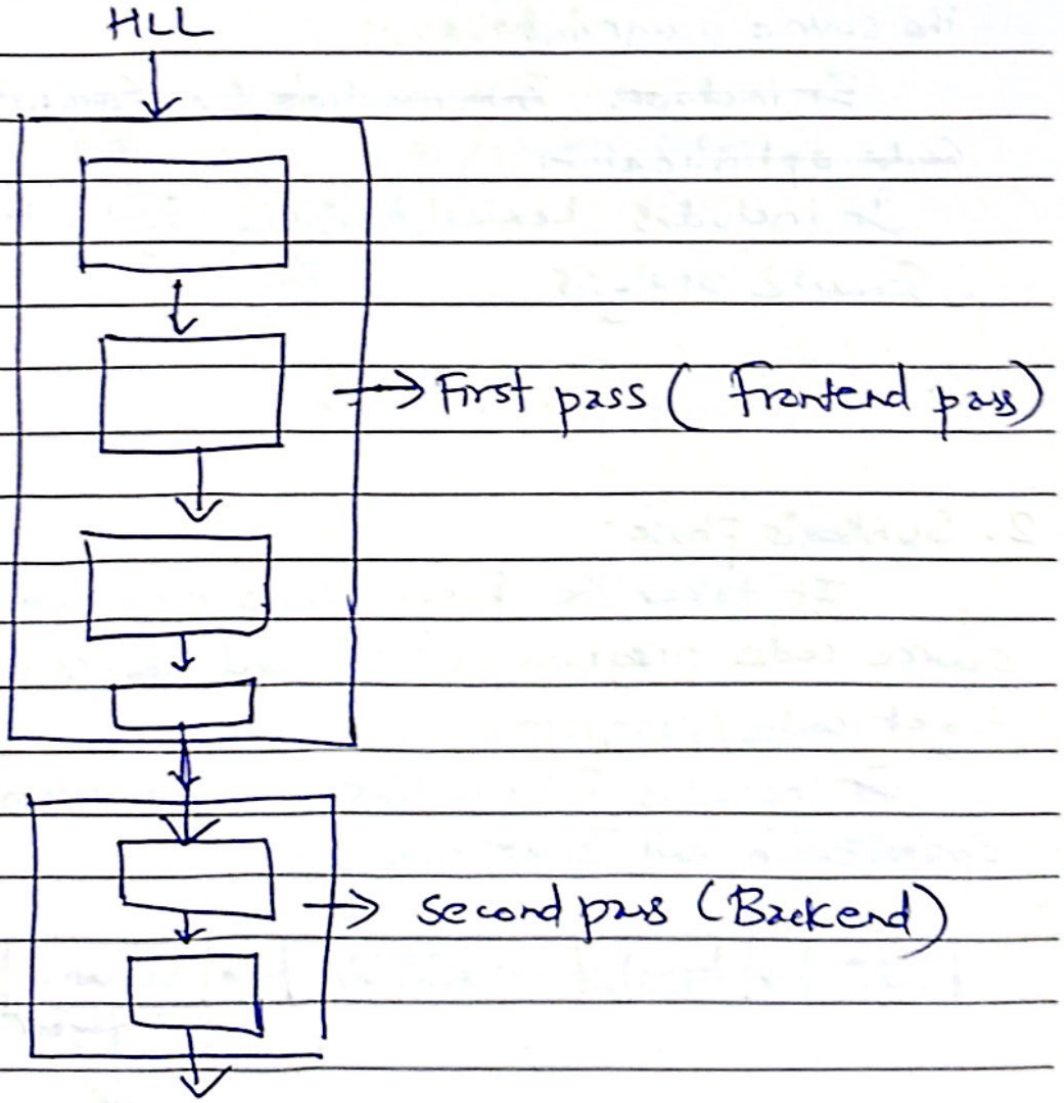


KHWOPA

S M T W TH F S Date .....

2) Multipass Compiler

It is a type of compiler that processes source code Multiple times (to convert HLL → Low level language) i/e to convert source code to target/object code

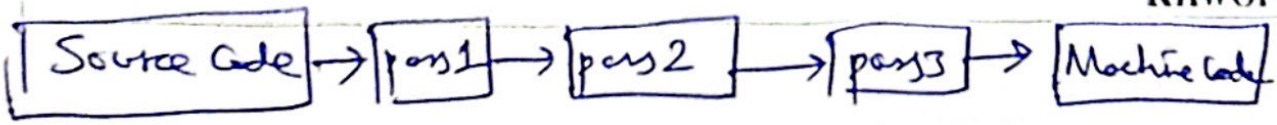


Advantage: Better optimization and Error Handling

Disadvantage: Slower Compilation

E.g: Modern C++ compilers (GCC, Clang)

KHWOPA



S M T W TH F S Date .....

Compilation Process

1. Analysis Phase:

It breaks down the source code/ program into small parts (tokens, patterns, lexeme) and checks syntax and semantics creates an Intermediate representation of the source program.

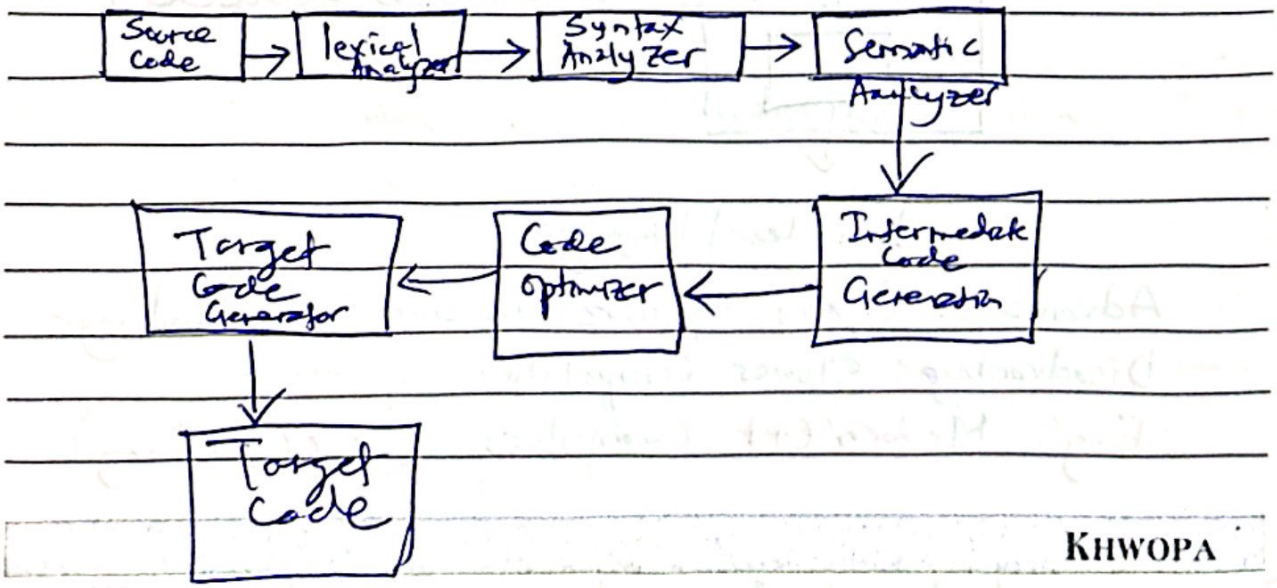
~~It includes Intermediate Code Generation (ICG), Code optimization~~

It includes Lexical Analysis, Syntax Analysis and Semantic analysis

2. Synthesis phase:

It takes the Intermediate representation of the source code program as input and creates the desired target code/program.

It includes Intermediate Code Generation, Code optimization and Target code



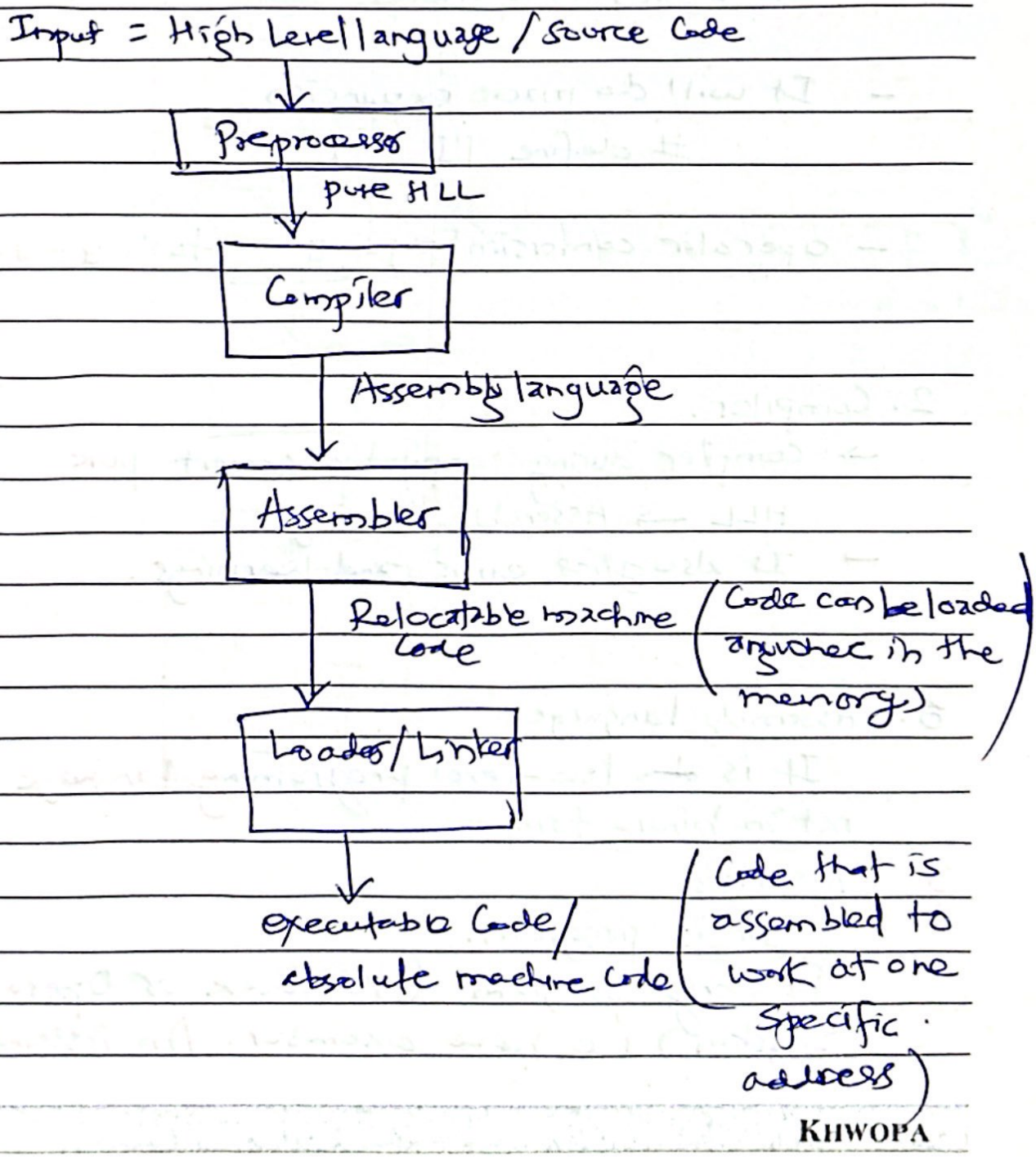
KHWOPA

S M T W TH F S Date .....

Language Processing Systems.

In Compiler Design we write the programs in High level language which is easy for us to understand.

These programs are fed into a series of tools and OS components to get the desired code that can be used by the machine code. This is called language.



KHWOPA

S M T W TH F S Date .....

1. Pre-processor

In Preprocessing, HLL Converted to Pure HLL
It handles macros, file inclusion and Conditional Compilation.

- Preprocessor removes the preprocessor directives (#include <stdio.h>) and will add the respective files i/e. file inclusion.

- It will do macro expansion
  #define PI 3.14

- operator conversion eg- a-- to a=a-1;

2. Compiler.

→ Compiler during compilation convert pure HLL → Assembly language.
→ It also gives errors and warnings.

3. Assembly language

It is low-level programming language. It is not in binary form.

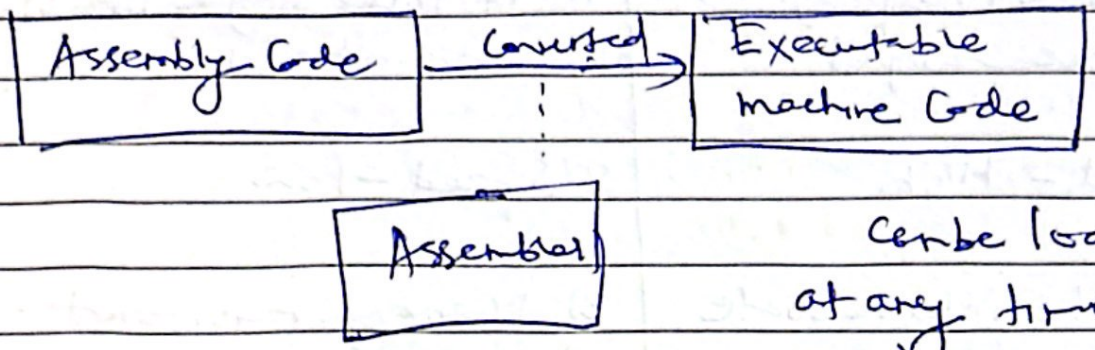
3. Assembler

It is program.
for every program (Hardware or Operating System) we have assembler. An Assembler

KHWOPA

S M T W TH F S Date .....

for one platforms will not work for another platform



Can be loaded at any time & can be run.

4. Loader & Linker.

Converts relocatable Code → Absolute machine Code

A Linker links different object files into a single file / executable file. and then loader loads that executable file in memory and executes it.

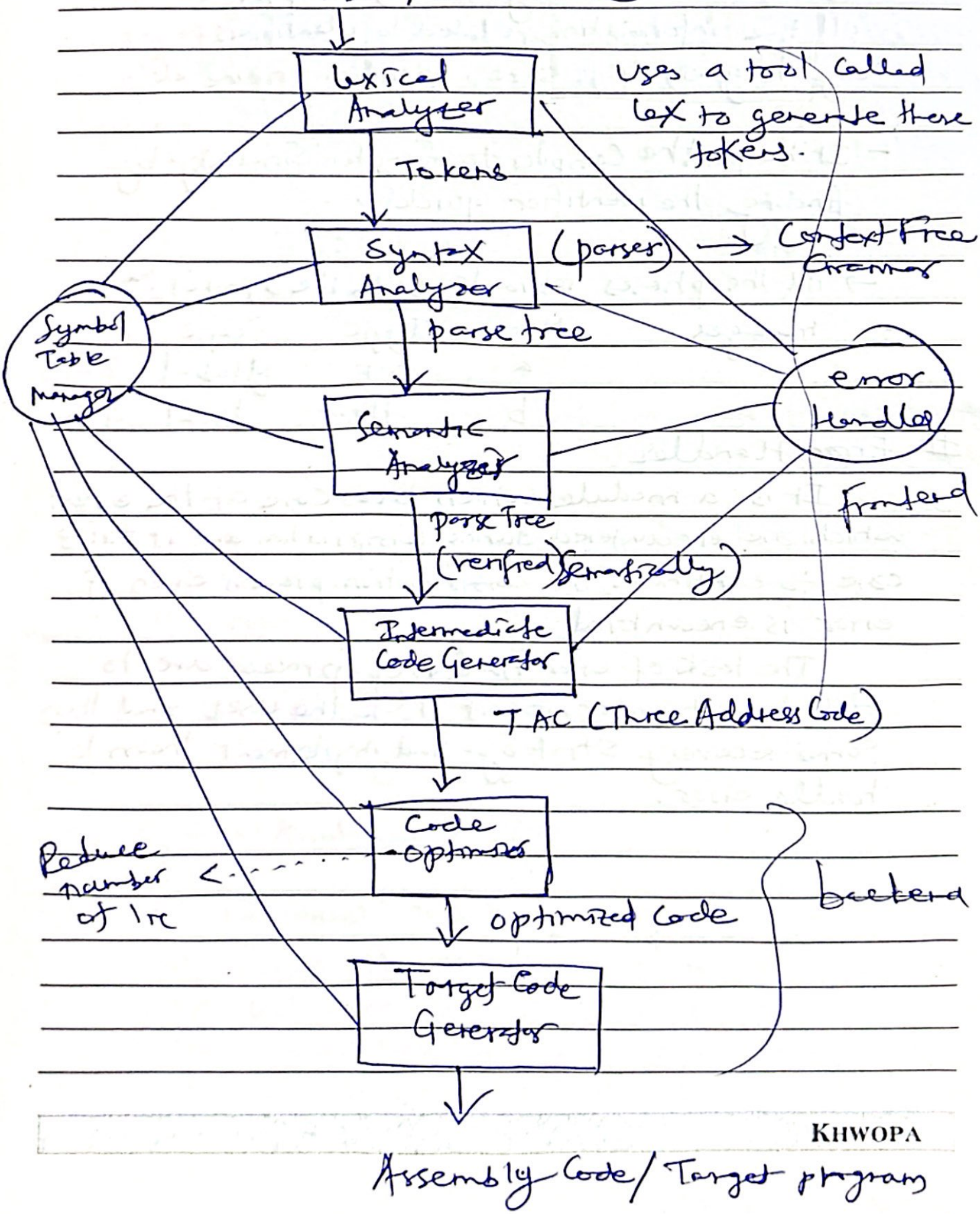
S M T W TH F S

Date .....

Compiler	Interpreter
1. Takes entire program at once as input	1. It takes single line of code at a time.
2) Speed - High	2. Speed - low
3) Generates intermediate object code. So, memory requirement is more.	3) Memory requirement is less because no intermediate code created.
4) Errors: All errors are displayed together	4) Errors: Continues translating the program until the first error is met, in which case it stops
5. Error detection is difficult	5. Error detection is easy
6) Compilers are larger in size.	6. Smaller in size
7) C, C++, Scala	7. Perl, python, Matlab Ruby

S M T W TH F S Date .....

Phases of a Compiler



KHWOPA

S M T WTH F S Date .....

Symbol Table Manager

→ Data structure being used by Compiler to store all the information related to identifiers.
eg its types, Scope, Size, location, name etc.

- It helps the Compiler to function Smoothly by finding the identifier quickly.

→ All the phases interacts with the Symbol table manager.

Name	Type	Scope	Size
a	int	global	2 Byte
b	float	local	4 Byte

Error Handler.

It is a module which takes care of the event which are encountered during Compilation and it takes care to continue the Compilation process even if error is encountered.

The task of error handling process are to detect each error, report it to the user and then some recovery strategy and implement them to handle error.

S M T W TH F S Date .....

Phases of Compiler

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation

} Analysis phase (Frontend)

5) Code optimizer
6) Target Code Generator

} Synthesis phase (Backend)

1. Lexical Analyzer.

- Reads the program and Converts characters into tokens using a tool called LEX tool.

- Tokens are defined by regular Expression which are understood by lexical Analyzer
- Removes white spaces, Comments, tabs

E.g. int a = 10;
Tokens: int, a, =, 10, ;

2) Syntax Analyzer / PARSER

→ Constructs the parse tree.

→ Takes the tokens one by one and uses Context free Grammar (CFG) to Construct the parse tree. If it is not possible to

SMIT MITH ES

Date .....

Construct the parse tree then input is ~~[illegible]~~ Syntatically incorrect and error message shown displayed. Using productions we can represent what the program actually is. The input has to be checked whether it is in the desired format or not.

→ Syntax errors can be detected by it if the given grammar is not accordingly.
→ So, it takes input as tokens and checks their Syntax Correctness.

eg. if (a>b) {...} → Valid Syntax.

3. Semantic Analyzer.
→ verifies the parse tree, whether it is meaningful or not.
→ It uses parse tree and information in the Symbol table to check the source program for semantic consistency with the language definition.
→ type checking, defining of the variables where, Label checking, flow control checking

eg. int a = "Hello"; → Error (type mismatch)

S M T W TH F S Date .....

4. Intermediate Code Generator.

→ Generates the intermediate code
- Three Address Code (TAC)
or
Abstract Syntax tree (AST)

5. Code optimizer

→ We get optimized code (i.e. our number of lines are reduced removing unnecessary code lines.

→ Code optimization phase attempts to improve the intermediate code so that it runs faster and consumes less resources

6. Target Code Generator

- Final phase of the Compiler which generates the target code.

S M T W TH F S Date .....

Sample Code & Compiler phases example.

```
#include <stdio.h>
int main() {
    int a=5, b=10;
    int sum = a+b;
    printf("sum: %d", sum);
    return 0;
}
```

→ Lexical Analysis:
Breaks codes to tokens
int, main, a, b, sum, printf, etc.

→ ~~Syntax~~ Analysis:
checks grammar.
eg. Correct use of int and printf.

→ Semantic Analysis: Ensures variables are declared before use.

→ Intermediate Code Generation
sum = a+b;

```
t1 = b
t2 = a+t1;
sum = t2;
```

KHWOPA

S M T W TH F S Date .....

- Code Optimization : Eliminates redundant operations.

→ Code Generation.

```
MOV AX, b
Add AX, a
Mov Sum, AX.
```

1. Lexical Analyzer

Function of lexical Analyzer

The main function of lexical Analyzer.

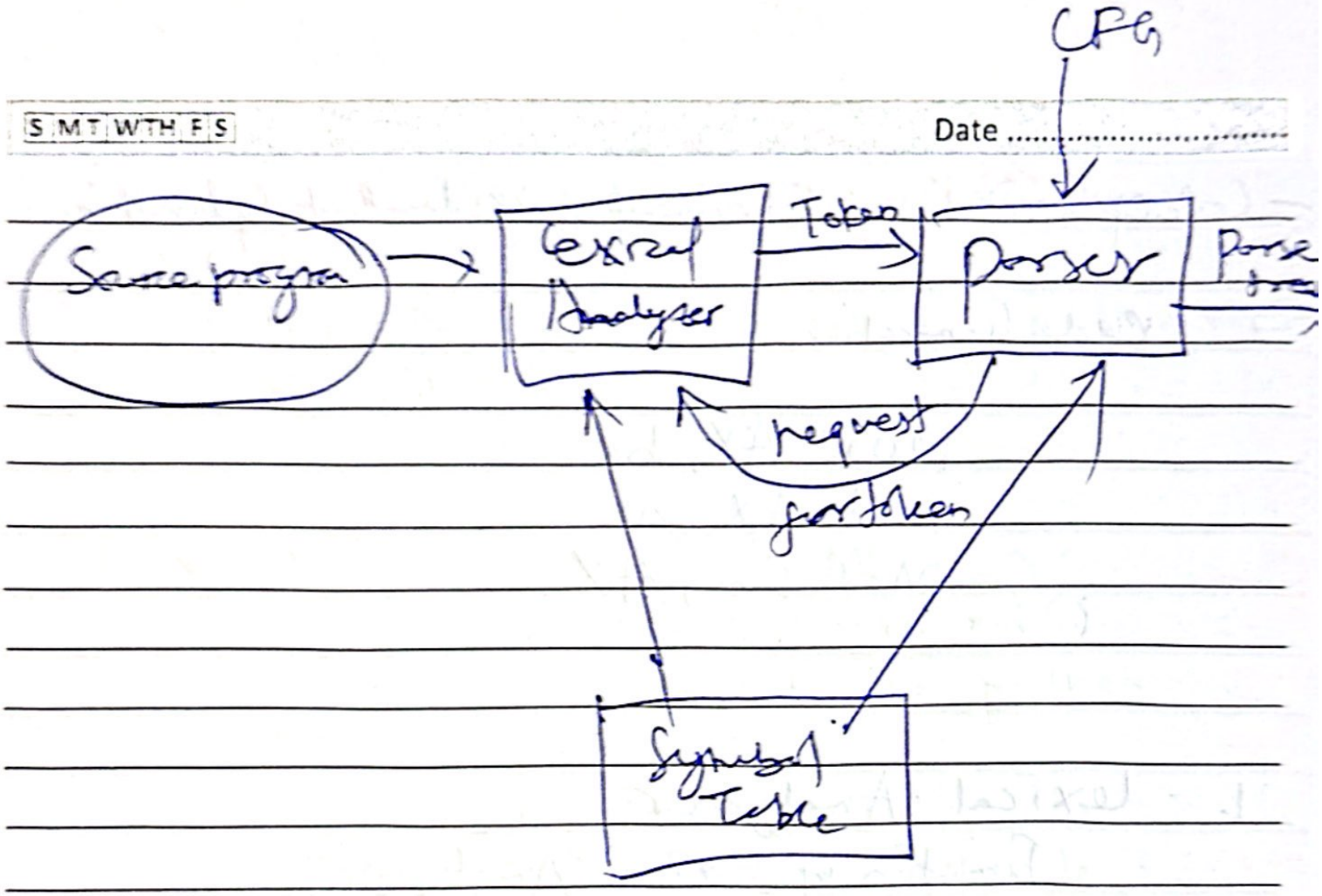
In lexical Analysis phase, Lexical Analyzer converts source program to streams of tokens.
- It is also called Scanning.

Function of lexical Analyzer

i). Reads the input program character by character and it produces a stream of token (and passes all the data to the Syntax Analyzer when it demands.

S M T W TH F S

Date .....



ii) Removes whitespace and Comments.

iii) Identifies keywords, Operators and identifiers

iv) lexical Analyser generates error and gives the line number of the error.

S M T W T H F S Date .....

E.g :

```
int x = 10;
```

Tokens

Keyword : int
Identifier : x
operator : =
Constant : 10
Punctuation : ;

if we pass a = b + c :

Token id = id + id ; (id = identifier)

where each id refers to its variable in Symbol Table.

TOKENS :

A token is a sequence of characters that can be treated as a unit/single logical entity.

Typically tokens are

- keywords (for, if, while, int ....)
- identifiers (variable name, function name)
- operator (+, -, ++)
- Constant
- punctuation separators , ; etc.

S M T W TH F S Date .....

PATTERN:

Pattern is a rule describing all those lexemes that can

LEXEME:

It is a sequence of characters in the source program that is matched by the pattern for a token.

or

a sequence of input characters that comprises a single token is called a lexeme.

eg, "float", "=", "f23", ";".

PATTERN:

Pattern is a rule describing all the lexemes that can represent a particular token in a source language.

Now we can define pattern as:

There are some predefined rules for every lexeme to be identified as a valid token. These rule are defined by grammar rules by means of patterns.

S M T W TH F S Date .....

E.g. int max (int i);

→ lexical Analyzer first read int and find it to be valid and accepts as token

→ max is read by it and found to be valid function name.

→ similarly (, int, i,) and ; are all valid tokens total tokens 7.

eg. int main()
{
// 2 variables declared below
int a, b;
a = 10;
return 0; Token = 18;
}

eg. printf("Name Engg"); Token = 5;

S M T W TH F S Date .....

lexical Analysis with DFA

lexical Analyzers use Deterministic finite Automata (DFA) to recognize patterns in source code with a finite number of states and transitions between them based on input characters.

A DFA recognizing identifiers

$([a-zA-Z\_][a-zA-Z0-9\_]^*$ Consisting

State q_0 (Start State): Transition to q_1 when receiving alphabet or _.

State q_1 (accept State): Accepts letters and digits but not special symbols.

processing of Count 1.

C → q_1, o → q_1, u → q_1, n → q_1, t → q_1
1 → q_1 (valid identifier)

int 3var = 5 error is detected because variable names cannot start with digit.

KHWOPA

Notes

Date .....

S M T W TH F S Date .....

Left Recursion.

A production of Grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its Left hand Side.

ie. $A \rightarrow A\alpha \mid \beta$

(i) Direct left recursion $B \rightarrow B\alpha$

(ii) Indirect left recursion
$S \rightarrow Aa$
$A \rightarrow Sm$

Why we remove Left Recursion.

Top Down Parser cannot accept the grammar having left recursion i.e. they don't allow Left recursive Grammar.

→ So, we have to remove left Recursion but preserve the language generated by grammar.

E.g in term of fxn.

```
A()                          A()
{                            {
  A();   Infinite loop         α;
  α;                           A();
}                            }
Left recursion               Right Recursion
```

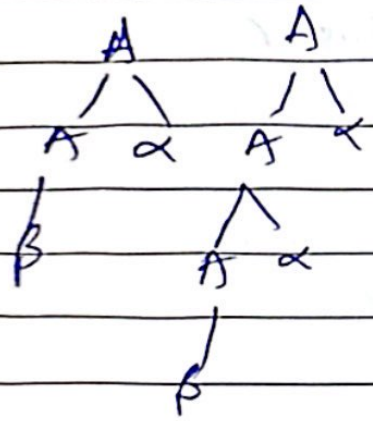
KHWOPA

S M T W TH F S Date .....

Recursion

LR

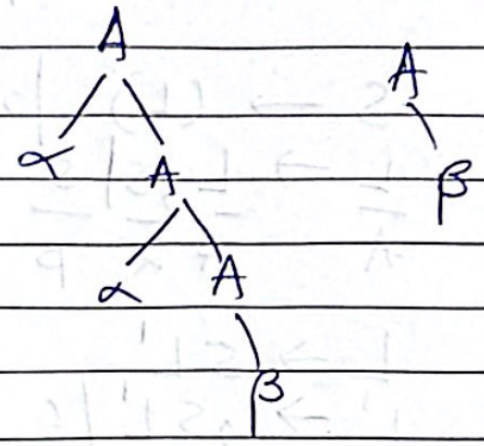
$A \to A\alpha \mid \beta$



yield (L) = $\beta\alpha^*$

RR

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



Language = $\alpha^*\beta$

Conversion

$A \to \beta A'$ ---> A will generate β followed by A'

$A' \to \alpha A' / \epsilon$ ---> Now A' will generate α^*

e.g. $E \to E+T / T$

A : $A \quad \alpha \quad \beta$

$E \to TE'$

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

KHWOPA

S M T W TH F S

Date .....

$A \to \beta A'$

$A' \to \alpha A' / \epsilon$

(e1) $S \to S\,0S1S / 01$

$A \quad A\,\alpha \quad \beta$

$S \to 0 / S'$

$S' \to 0S1S\,S' / \epsilon$

(e2) $S \to (L) / x$ no left recur

$L \to L, S / S$

$A \quad A\,\alpha \quad \beta$

$L \to SL'$

$L' \to ,SL' / \epsilon$

(e3) $A \to A\alpha_1 / A\alpha_2 / A\alpha_3 / \ldots / \beta_1 / \beta_2 / \beta_3$

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

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

Summary (1) Grammar

Ambiguous | Un

2 parse tree

(no parser work for it)

(2) Grammar

LR | LR

X

because Top Down parser will not accept it because will fall in infinite loop.

KHWOPA

S M T WTH F S

Date .....

(iv) $E \to E+T \mid T$

$T \to T*f \mid f$

$f \to id$

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

$A \to \beta A'$

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

1st production

$E \to TE'$

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

Saying for 2nd production

$T \to T * F \mid F$

$T \to FT'$

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

3rd production $F \to id$ ok

Final Answer ~~$E \to TE' \; / \; +TE' \mid \epsilon$~~

$E \to TE'$

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

$T \to FT'$

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

$F \to id$

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

$L \to L, S \mid S$

$L \to SL'$

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

(vi) $S \to Aa$

$A \to Sb \mid C$

$A \to Aab \mid C$

$A \to CA'$

$A' \to abA' \mid \epsilon$

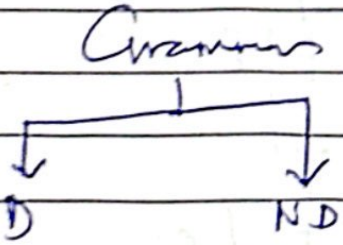
KHWOPA

S M T W TH F S Date .....

LEFT FACTORING

Sometimes it is not clear which production to choose to expand a non-terminal because multiple productions begin with the same terminal / non terminal lookahead.

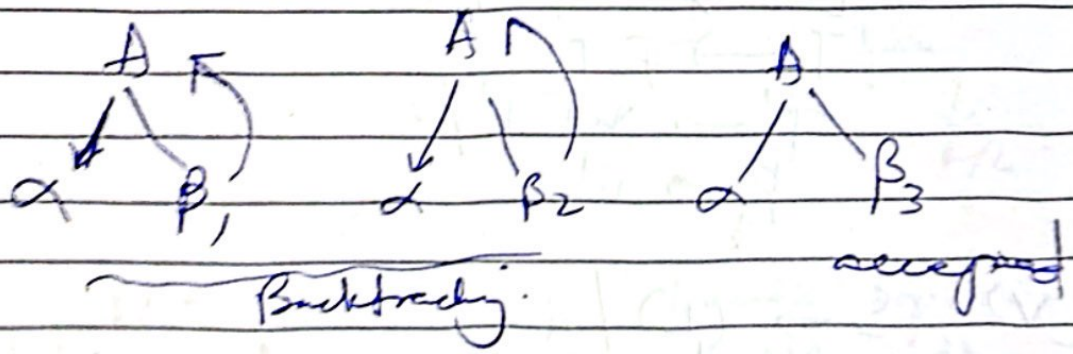
This type of Grammar → Non Deterministic grammar or Grammar containing left factoring



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

Common prefixes.

Suppose we have to accept $\alpha\beta_3$



This is happening because of the common prefixes or (Context)

S M T W TH F S Date .....

One or more productions on the RHS are having something common in the prefixes (context)

This is called common prefix problem or NDA deterministics grammar.

Backtracking happens ?

→ we are making our decision only after seeing prefix. (α in example)

i.e which production to choose without seeing full input.

If input is $\alpha\beta_3$

Non-Deterministic Grammar

$$A \to \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3 \quad (NDG)$$

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

} we have postponed the decision making.

The procedure we used to convert Non-Deterministic grammar → Deterministic Grammar is called left factoring.

eg. $S \to iEtSeS / iEtS / a$
$E \to b$

$S \to iEtS\,S'$
$S' \to eS / \epsilon / a$
$E \to b$

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

$S \to aS' / b$
$S' \to SSbS / aSb / bb$
$S' \to SS'' / bb$
$S'' \to SbS / aSb$

KHWOPA

S M T W TH F S

Date .....

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

$S \rightarrow bSS' \mid a$

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

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

$S'' \rightarrow aS \mid Sb$

Parser

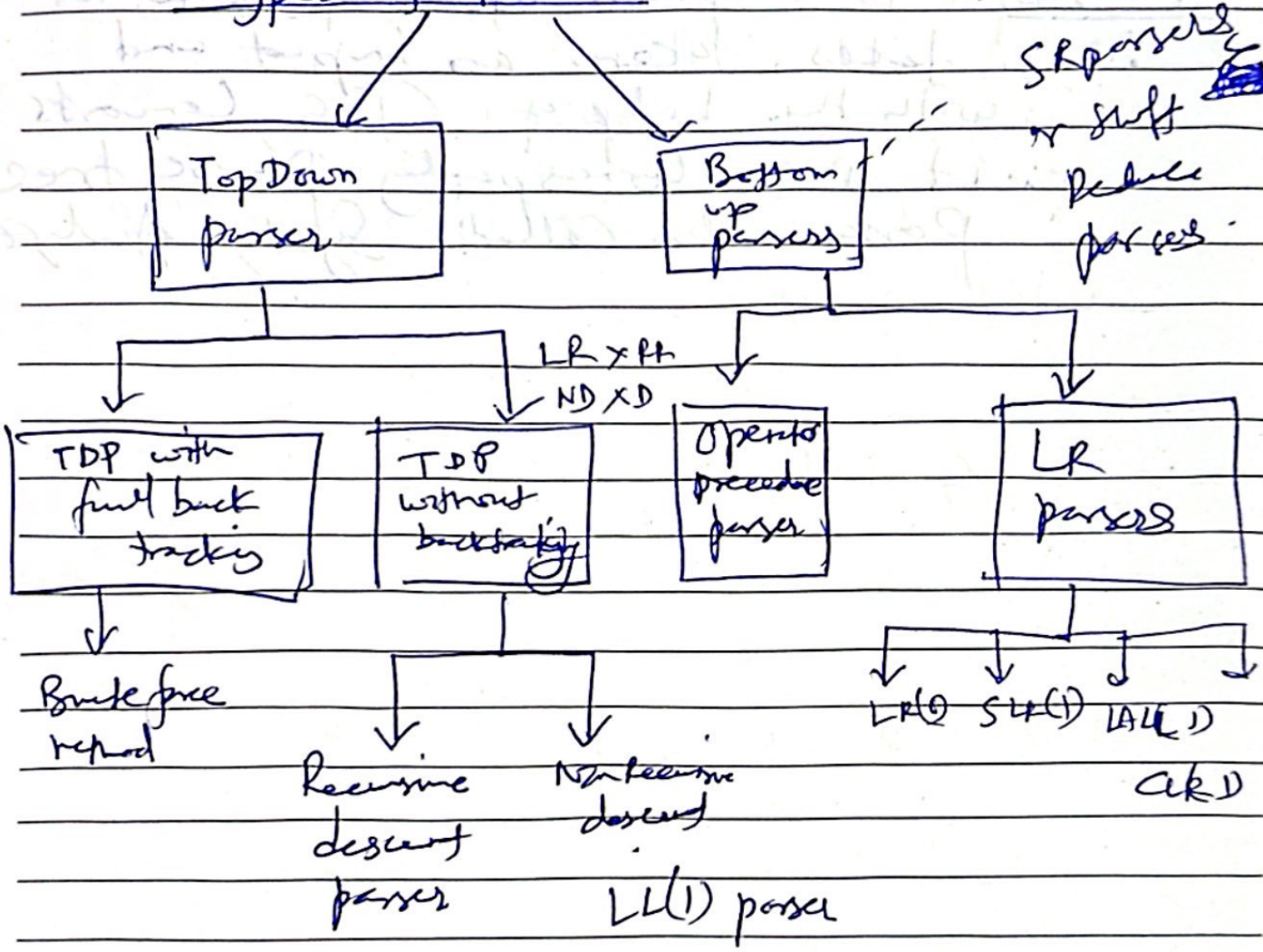
S M T W TH F S

Date .....

Parser: It is phase of compiler which takes tokens as input and with the help of CFG, converts it into corresponding parse tree. Parser also called Syntax Analyzer.

S M T W TH F S Date .....

Types of Parsers



we will have some recursive

LL(1) parsers → look aheads (number of Symbol)

L - Left → right

L - Left most derivation

abcd

LR parser

L - Left → right (Scan)

R → RMD using reverse of RMD.

KHWOPA

S M T W TH F S

Date .....

Variety of Grammar	TD parser	Bottom up parser
→ Ambiguous Grammar	X	X except operator precedence parser
→ Unambiguous	✓	✓
3) LR	X	✓
4) RR	✓	✓
5) ND	X	✓
6 Deterministic	✓	✓

KHWOPA

S M T W TH F S Date .....

Top Down Parser

It generates parse tree for the given input string with the help of grammar productions by expanding the non-terminals i.e.
It starts from the Start Symbol and ends on terminals.

- It uses LMD (Left Most Derivation)

Bottomup Parser

It generates the parse tree for the given input string with the help of grammar production by compressing the non-terminals i/e It starts from the terminals and ends on the Start Symbol.

- It uses reverse of Right Most Derivation

S M T W TH F S | Date .....

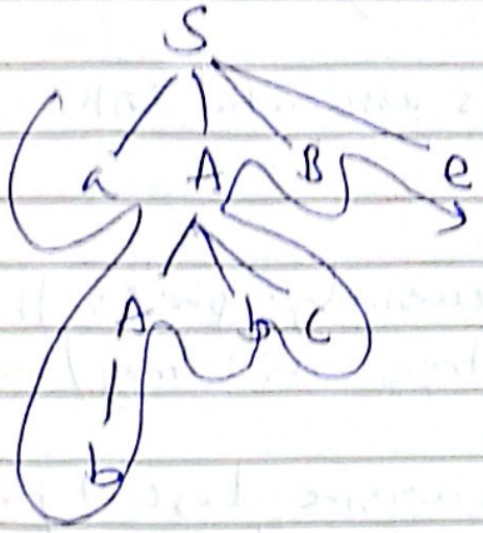
$S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$\omega \rightarrow abbcde.$

TDP:



LMD:

$S \rightarrow a\underline{A}Be$

$\rightarrow a\underline{A}bcBe$

$\rightarrow abbc\underline{B}e$

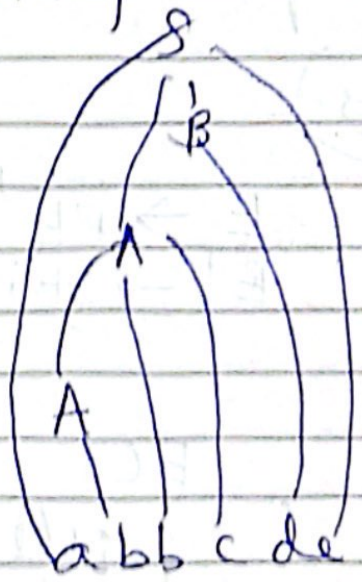
$\rightarrow abbcde$

Top down parsing → At every point we have to decide what is the next production we should use.

BUP: Bottom up parsing.

main decision we have to make is

" when to reduce the given derivation"



RMD: $S \rightarrow aA\underline{B}e$

$\rightarrow aAde$

$\rightarrow aAbcde$

$\rightarrow abbcde$

KHWOPA

S M T W TH F S Date .....

Recursive Decent Parser.

- It is top-down parser technique that constructs the parse tree from top and the input is read from left-right.

- A procedure is associated with each non-terminal of the grammar.

- This technique recursively parses the input to make a parse tree which may/may not require (backtracking).

- A form of recursive descent parser that doesn't require any backtracking is called predictive parser.

Eg.

$E \rightarrow i\,E'$

$E' \rightarrow +i\,E' \mid \in$

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

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

```
main()
{ E();
  if(l=='$')
    printf("success");
}
```

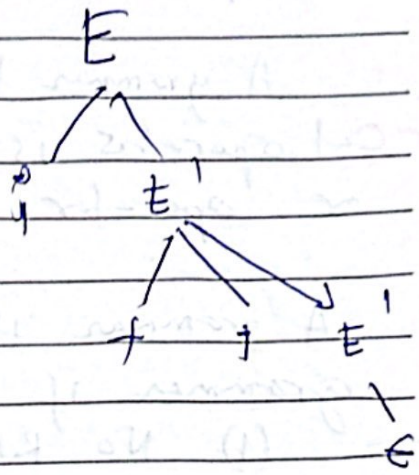
```
match(char t)
{ if(l==t)
    l=getchar();
  else
    printf("error");
}
```

In lookahead

S M T W T H F S

Date .....

$\hat{i} + \hat{j}$ $ erdq sfrj.



S M T W TH F S

Date .....

Operator Grammar

A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar.

A grammar is said to be operator precedence grammar if it has two properties.

(1) No R.H.S of any production has ϵ.

(2) No two non-terminals are adjacent or R.H.S.

eg. $E \rightarrow E+E \mid E*E \mid id.$ / Operator grammar.

eg. $S \rightarrow SAS \mid a$
$A \rightarrow bSb \mid b$ } Not operator grammar

we will Convert it into operator gram

$S \rightarrow SbSbS \mid a \mid SbS$

$A \rightarrow bSb \mid b$ Not required.

S M T W TH F S Date .....

There are 3 operator precedence relations:

1) $a \doteq b$ means a has higher precedence than the terminal b. or "a takes precedence over b".

2) $a \lessdot b$ means a 'yields precedence to" b or a has lowest precedence than terminal b.

3) $a \doteq b$ means a has same precedence as b.

Operator Precedence Parser

→ Bottom up parser that interprets an operator grammar.
→ This parser is only used for operator grammar.
→ Ambiguous grammar are not allowed in any parser except operator precedence parser.