FRANKLIN'S LECTURES

COMPILER DESIGN

Module 1 Part 1

CST302

SYLLABUS

Module - 1 (Introduction to compilers and lexical analysis)

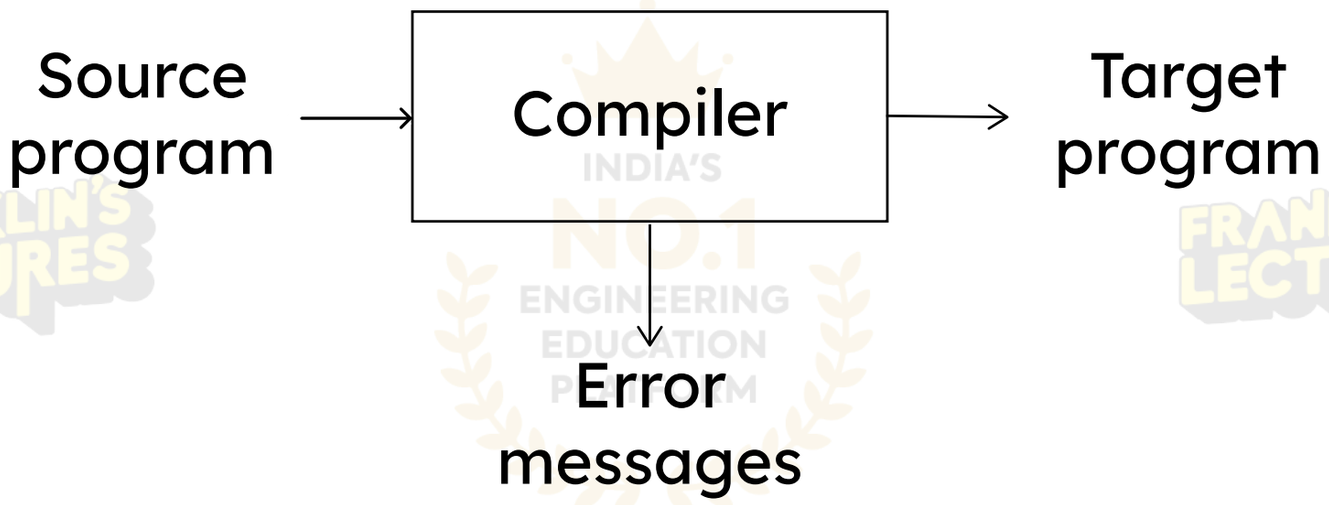
Analysis of the source program - Analysis and synthesis phases, Phases of a compiler. Compiler writing tools. Bootstrapping. Lexical Analysis - Role of Lexical Analyser, Input Buffering,Specification of Tokens, Recognition of Tokens.

PRACTICAL APPLICATIONS

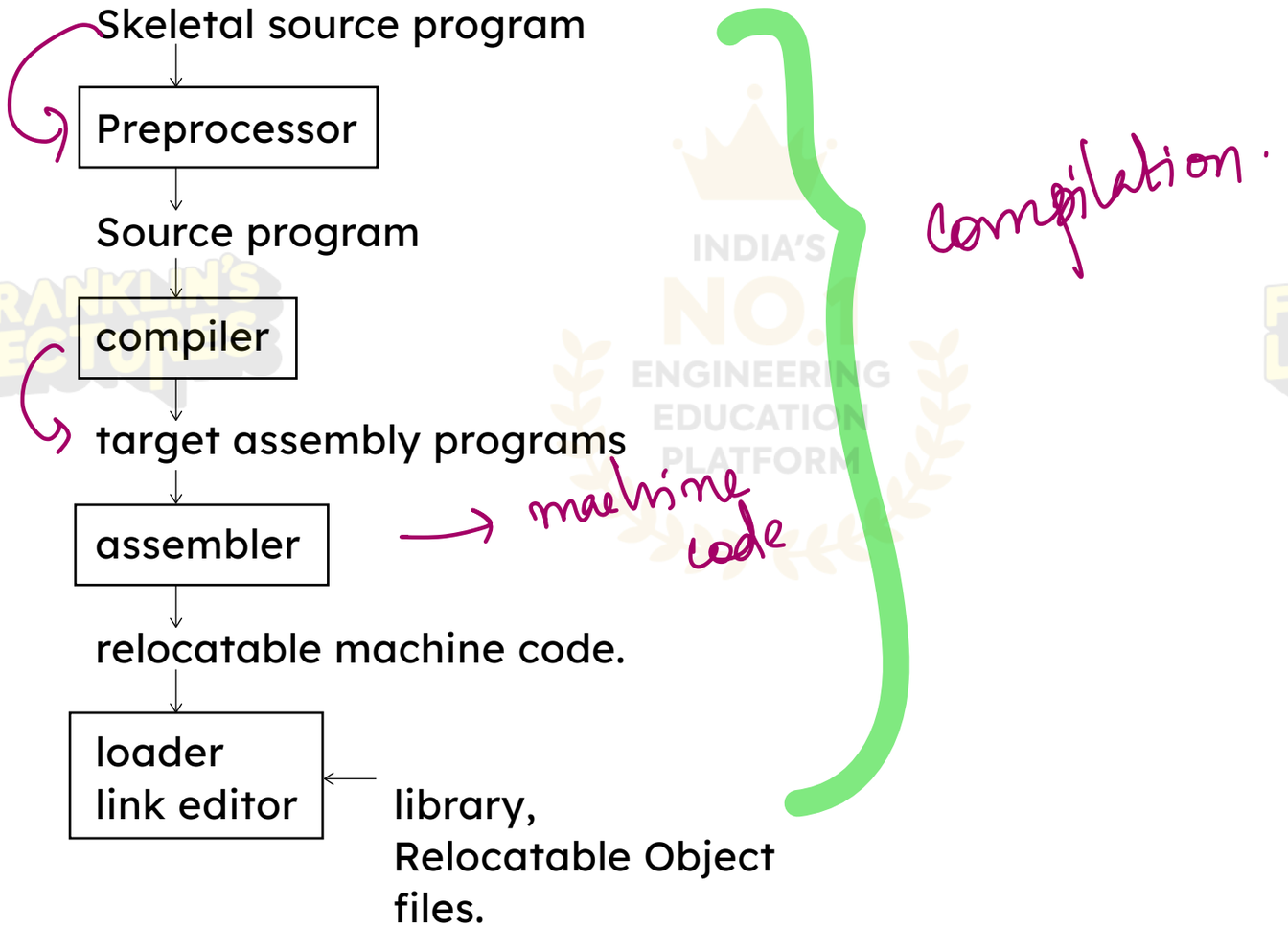
- Syntax checking in programming tools
- Syntax highlighting in editors
- Detecting invalid characters in code.

COMPILER

- A compiler is a program that reads a program written in one language-the Source language, and translates it into an equivalent program in another language-the target language. During this translation process, the compiler reports to its user the presence of errors in the source program.
- In other words a compiler is a system software that converts source program written in high level language Into a target program in low level language.


Source program
Compiler
Target program
Error messages

A LANGUAGE PROCESSING SYSTEMS



- In addition to a compiler, several other programs may be required to create an executable target program. A source Program may be divided into modules stored in separate files. The task of collecting the source program is done by a distinct program called preprocessor. The preprocessor may also expand short hands called macros, into source language statements
- The above figure shows a typical compilation. The target program created by the compiler may require further processing before it can be run. The compiler creates assembly code that is translated by an assembler into machine code and then linked together with some library routines into the code that actually runs on the machine.

Analysis of The Source program

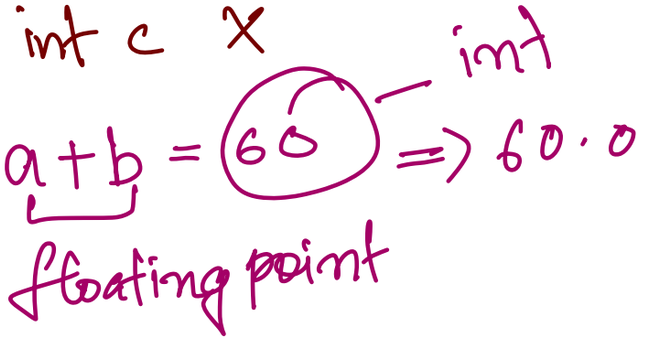
The analysis part breaks up the Source program into constituent pieces, and creates an intermediate representation In compiling analysis phase consists of three phases

1) Linear Analysis - Lexical Analysis / Scanning
2) Hierarchical Analysis - Syntax
3) Semantic analysis - Type checking

c. Pooqm

int c; ✓

int c x



phases of compiler C 14 mark (7|8|9|10)

compiler operates in phases, each of which transforms the source program from one representation to another. Different phases of compiler are,

ANALYSIS — source prgm → parts → grammatic structure(check)

- Lexical Analysis
- Syntactic Analysis
- Semantic Analysis

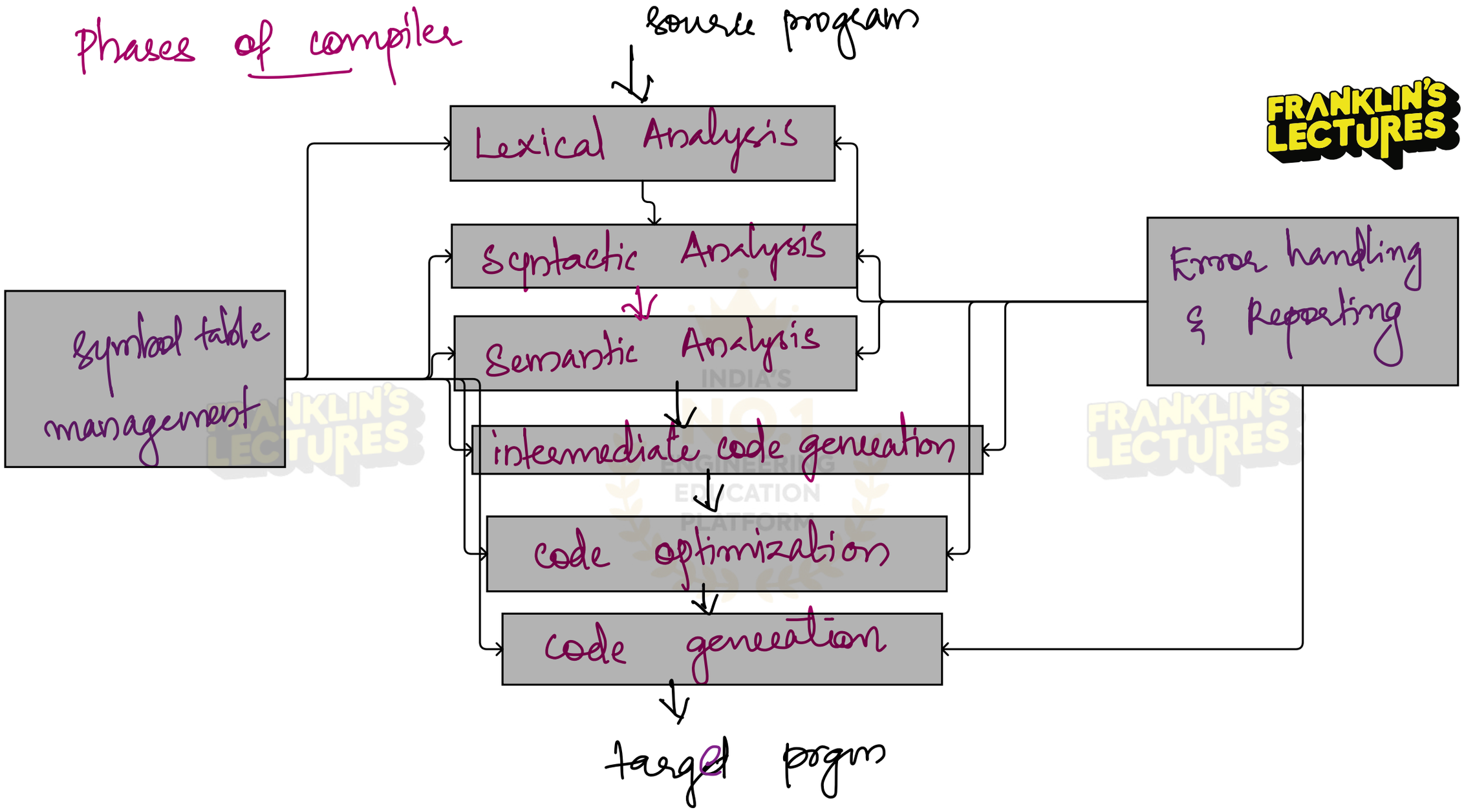
structure – ref – collect info abt source prgm & store in symbol table

SYNTHESIS

- Intermediate code generation
- Code optimization
- Code generation

source-prgm → target [code/prgm] machine code

Phases of compiler



Two other activities with Phases of compiles are,

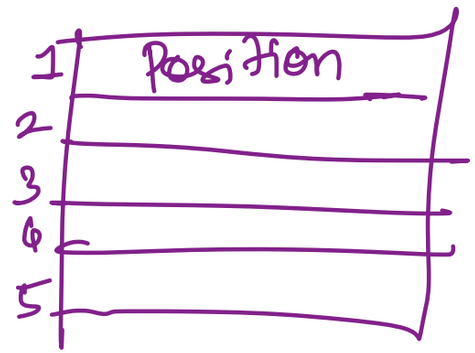
- Symbol Table management
- Error handling.

1) Lexical Analysis

stmnts / characters → tokens

- In a compiler linear analysis is called lexical analysis or scanning.
  eg:- in lexical analysis, the characters. in the assignment statement
- position = initial + rate * 60 can be grouped into the following tokens.

id1 id2 id3



i) identifier position

ii) The assignment symbol ":="

iii) The identifier initial

iv) The + sign

I) The identifier rate

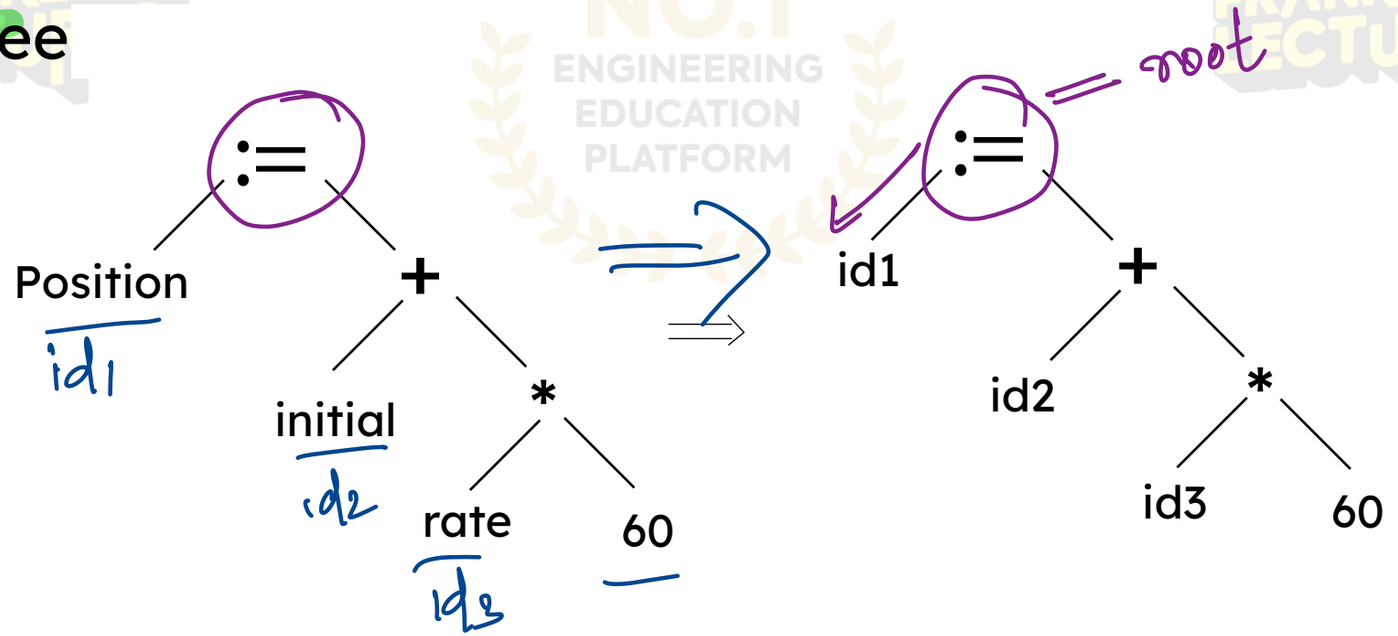
vi) The * sign

vii) the number 60.

→ the white spaces (blank space, tab, new line etc) would be eliminated during lexical analysis phase

2) Syntax Analysis

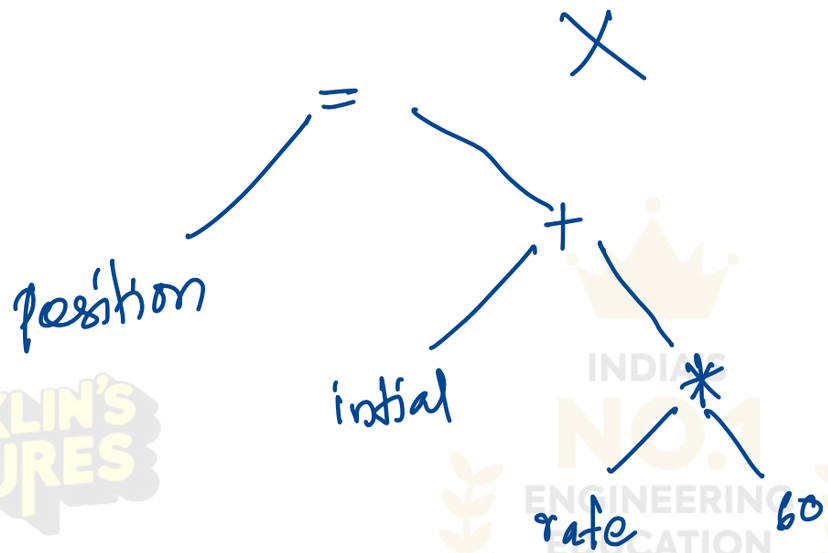
- Syntax analysis is called parsing or hierarchical analysis.
- It involves grouping of tokens of the source program into grammatical phrases.
- Usually grammatical phrases are represented by parse trees or syntax tree

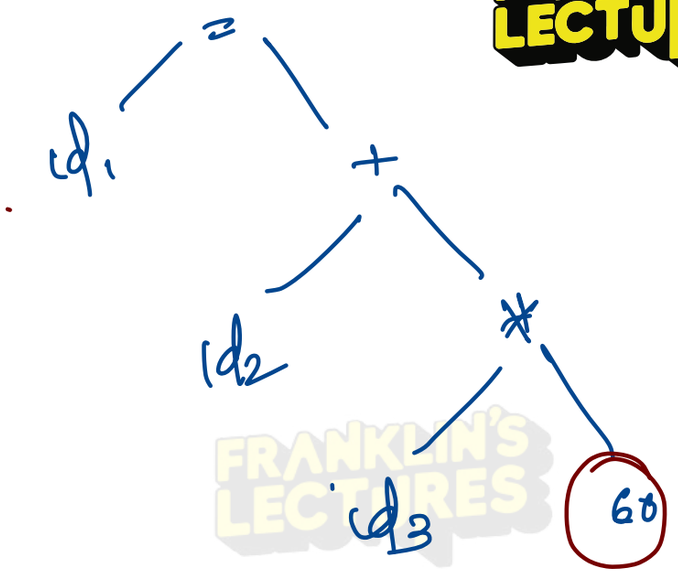


3) Sematic Analysis

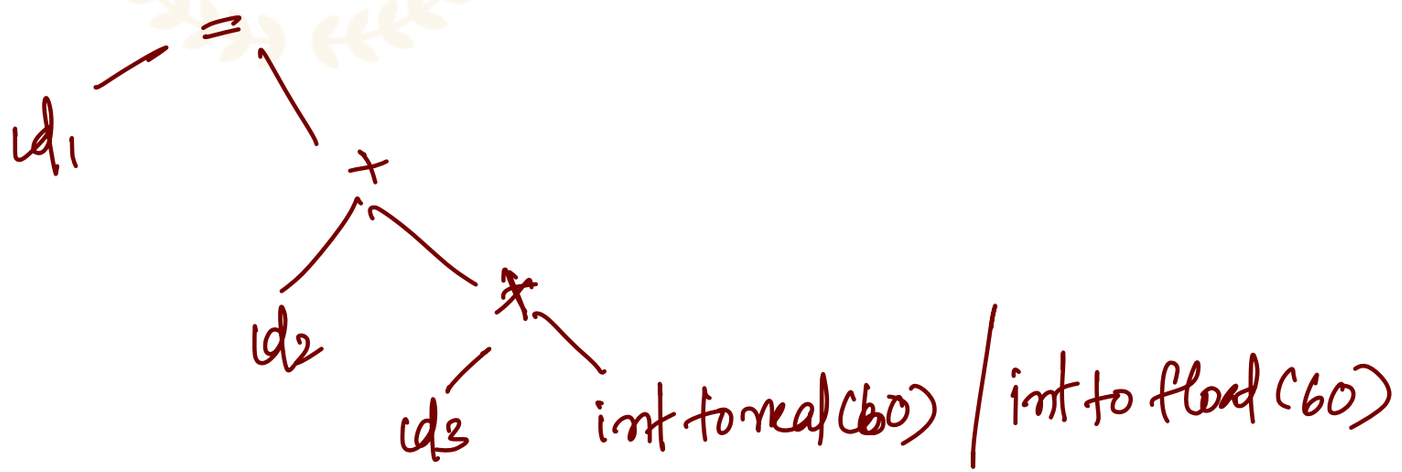
- This phase checks the source program for semantic errors. It uses the heirachieal structure determined by by the syntactic analysis phase to identify operators and operands of expressions and Statements.
- An important component of semantic analysis is type checking. Here the compiler checks that each operators has operands that are permitted by the source language specification.
- eg:- many programming language definition require a compiler to report an error when an a real number is used to index an array.

Position = initial + rate * 60 → position, initial, rate ⇒ floating point

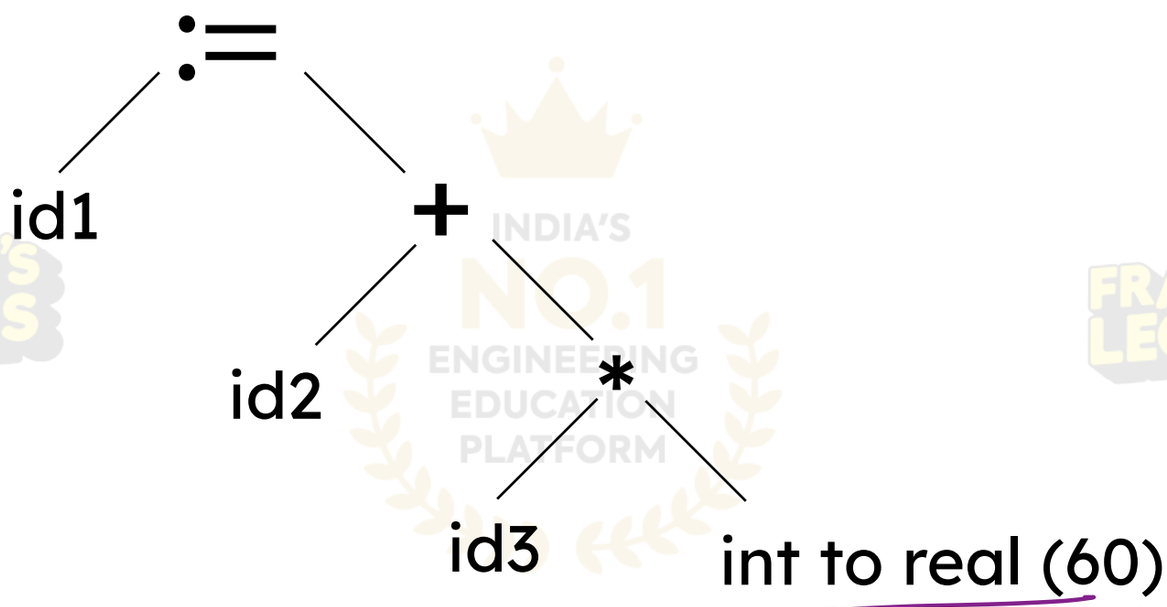




⇒



Tree after semantic analysis



4. Intermediate code generation

- After syntax and semantic analysis. some compilers generate an intermediate representations of the source program. This intermediate representations should have two properties,
- It should be easy to produce.
- It should be easy to translate is into the target program.
- Three kinds of intermediate representation are,

1) Syntax tree representation
2) postfix notations
3) 3 address code representation.

$id_1 = id_2 + id_3 * 60$

- 3 address code consists of sequence. of instructions each of which has atmost 3 operands. The source program, position:= initial + rate * 60 might appear in 3 address Code as,

- temp 1= Int to real (60)
- temp2: = id3 * temp1
- temp 3: = id2 + temp2.
- id1 : = temp 3.

temp 1 = int to real (60)
temp 2 = id3 * temp1
temp3 = id2 + temp2
id_1 = temp 3

- This intermediate form has several Properties,

- Each 3 address instruction has atmost two operators including assignment operator.
- The compiler must generate a temporary name to hold the value computed by each instruction
- Some 3 address instruction has fewer than three operands.
- eg:- temp1:= int to real (60)
- id1 : = temp 3.

$id_1 = id_2 + id_3 * 60$

floating point.

intermediate code

temp1 = int to Real(60) ⇒ 60.0

temp2 = id_3 * temp1 ⇒ temp1 = id_3 * 60.0

temp3 = id_2 + temp2 ⇒ id_1 = id_2 + temp1

id_1 = temp3

MOV source, destⁿ

R_1 R_2

```
MOVF  id3, R1
MULF  #60.0, R1
MOVF  id2, R2
ADDF  R1, R2
MOVF  R2, id1
```

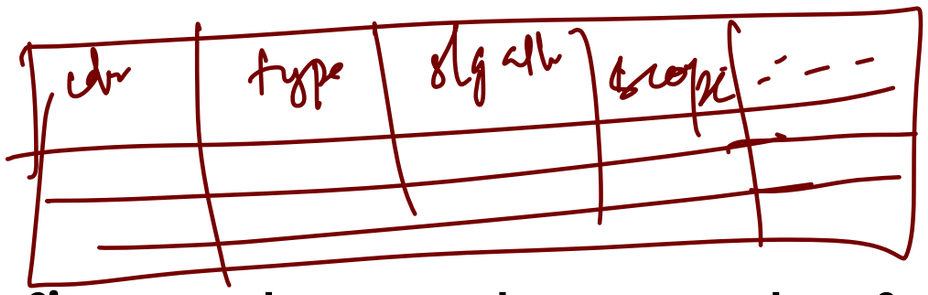
5) code optimizations

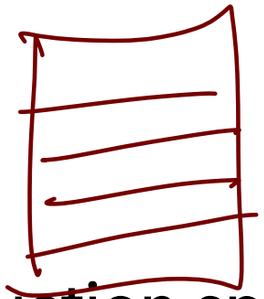
- The code optimization Phase attempts to improve the intermediate code so that faster running machine code will result.
- For eg: the intermediate code generated in intermediate code generation phase can be optimized by, as,
- temp1:=id3 * 60.0.
- id1 := id2+ temp1

6) Code generations

- The final phase of the compiler is the generation of the target code consisting normally of relocatable machine code or assembly code. Memory locations are selected for each of the variables used by the program.
- Then intermediate instruction are each translated into a sequence of machine instructions that performs the same task.
- For eg:- using registers 1&2 the translation of the code 1 might become
- MOVF id3, R2.
- MULF #60.0, R2.
- MOVF id2, R1
- ADDF R2, R1
- MOVF R1, id1.

source, dest





- The first and second operands of each instruction specify a source and destination respectively. The 'F' in each instruction tells that the instruction deal with floating point numbers. The #" sign signify that 60. 0 is to be treated as a constant

Symbol Table Management

- An essential function of compiler is to record the identifier used in the source program and collect Information about various attributes of each identifier. The attributes may provide information about the storage allocated for an Identitiers, its type, its scope and in the case of procedure name, such things as the number and type of its arguments,

- The method of passing arguments and the type of return value if any.
- A symbol table is a data structure containing a record for each identifier. containing with fields for attributes of the identifier. This data structure allows us to find record for each identifier quickly and to store or retrieve data from the record quickly.
- when an identifier in the source program is detected by the lexical analyzer, the identifier is entered is the symbol table However the attributes of an identifier cannot normally be determined during lexical analysis. The remaining Phases enter information about identifiers into the Symbol table and then use this information in various ways.

Error detection and Reporting

- Each phase can encounter errors. The Syntax and semantic analysis Phases usually handles large fraction of errors detectable by the compilers.
- The lexical phase can detect errors when the characters remaining in the input do not forms any tokens of the language.
  eg:- Misspelling of keywords
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
  eg:- Semicolon missing.

- During semantic analysis phase compiler try to detect constructs that have the right syntactic structure but no meaning to the operations involved. For eg: if we try to add two numbers one which is the name of an array & the other name of a procedure.

Grouping of the phases

- A number of phases can be grouped into a pass so that their activities can be interleaved together in a pass.
- Usually the front end consists of phases that depend upon the source language and are independent of the target language. It includes lexical analysis, syntactic analysis, Semantic analysis, generation of intermediate code and creation of symbol tables.
- Some amount of code optimization and error handling can also be included in this phase.
- The back end includes the phases of compilation that depends on the target machine. It includes code optimization, Symbol table operations error handling and code generation.

- The following are the effects of reducing the number of passes.

1. When the number of passes is reduced, the time taken to read & write intermediate files to or from the disk can be reduced.
2. On reducing the number of passes, the entire information of the pass has to be Stored in the temporary memory. This increases the memory space needed to Store the information.
3. When phases have more coupling among them, it is advantageous to group them together. For eg:- lexical analysis and Syntactive analysis are grouped together as the lexical analyzer can fetch the token as and when requested by the Syntax analyzer.

4. Related Phases should only be grouped together

(a) Code generation cannot be performed until an intermediate representation is available. So it cannot be grouped with the syntax analysis phase.

(b) Intermediate code and target code generation can be merged together.

writing (essay)

Compiler Construction Tools

- Compiler construction tools are the tools that have been created for automatic design of specific compiler components. These tools use specialized languages and algorithms The following is a list of compiler construction tools.

1) Scanner generators

i/p : Regular expsn of tokens of lang
o/p : lexical analyzer

- They generate lexical analyzers automatically from the language specifications written using regular expressions. It generates finite automata to recognize the regular expression. An example of this tool is lex.

→2) parser generators

- They produce syntax analyzers from context Free grammars. Many parses generator utilize powerful parsing algorithms that are too complex to be carried out by hand. An example for this tool is YACC. (Yet another compiler compiler)

i/p : Grammatic description of program lang

o/p : Syntax analyzer

3) Syntax directed translation Engine

i/p : parse tree
o/p : intermediate code

- These engine have routines to traverse the parse tree and produce intermediate code, one or more translations are associated with each node of the parse tree.

4) Automatic code generators

- Those tools convert the intermediate language into machine language for the target machine using a collection of rules. Template matching Process is used. An intermediate language statement is replaced by its equivalent machine language statement.

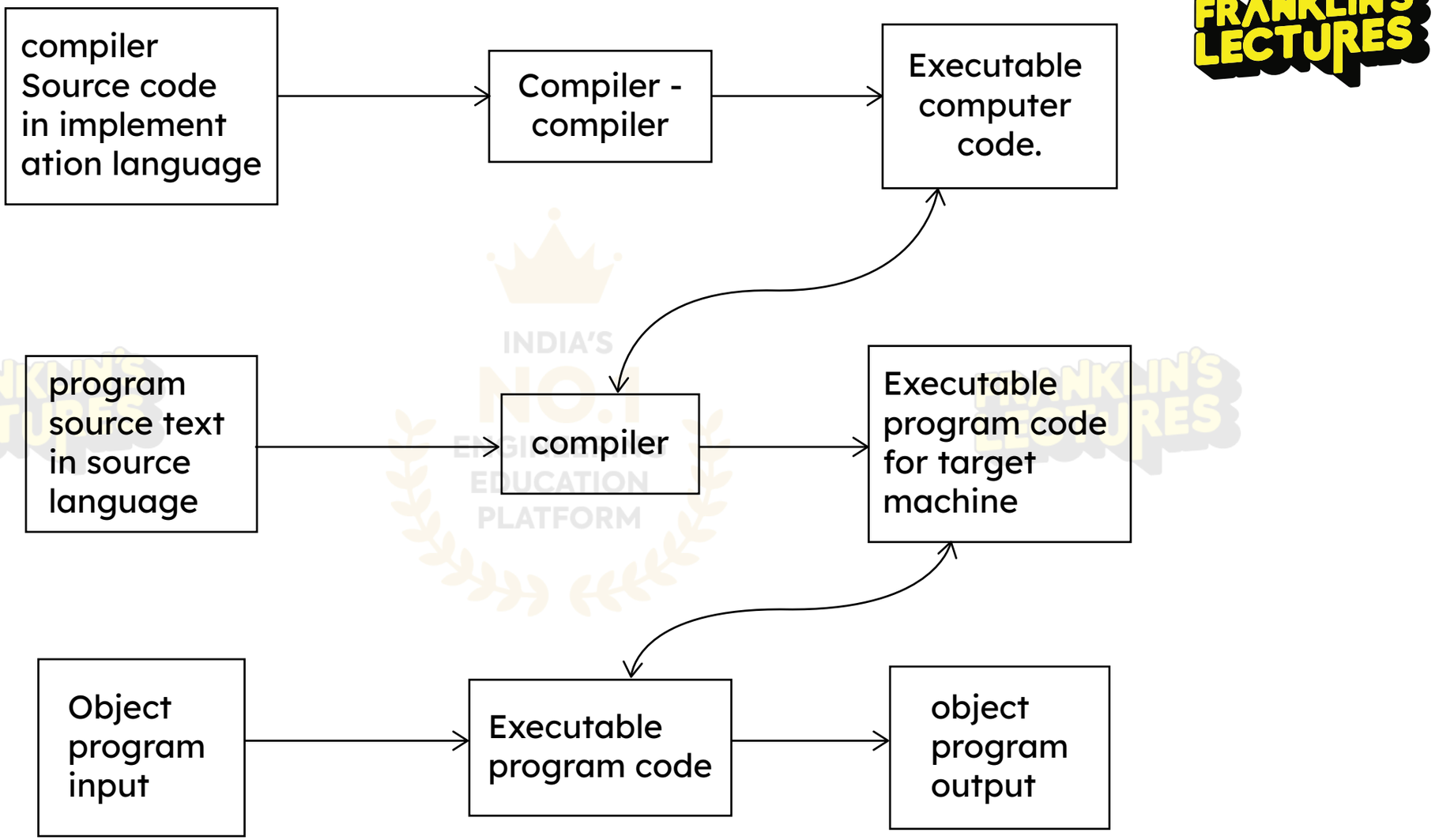
i/p : intermediate lang
o/p : machine lang.

5) Data Flow engines

- It is used in code optimization. These tools perform good code optimization using "data-flow analysis", the gathering of information about how values are transmitted from one part of a programs to each other parts.

Bootstrapping Compiler

- To obtain the compiler, its source text serves as an input to another compiler which produces an executable code for the source text. This is called as the executable file of the compiler. Then the source program input by the user is compiled by this compiler to produce the object code, which is loaded into memory and executed. The process of compiling and running a compiler is represented in the following figure.


compiler
Source code
in implement
ation language
Compiler -
compiler
Executable
computer
code.
program
source text
in source
language
compiler
Executable
program code
for target
machine
Object
program
input
Executable
program code
object
program
output

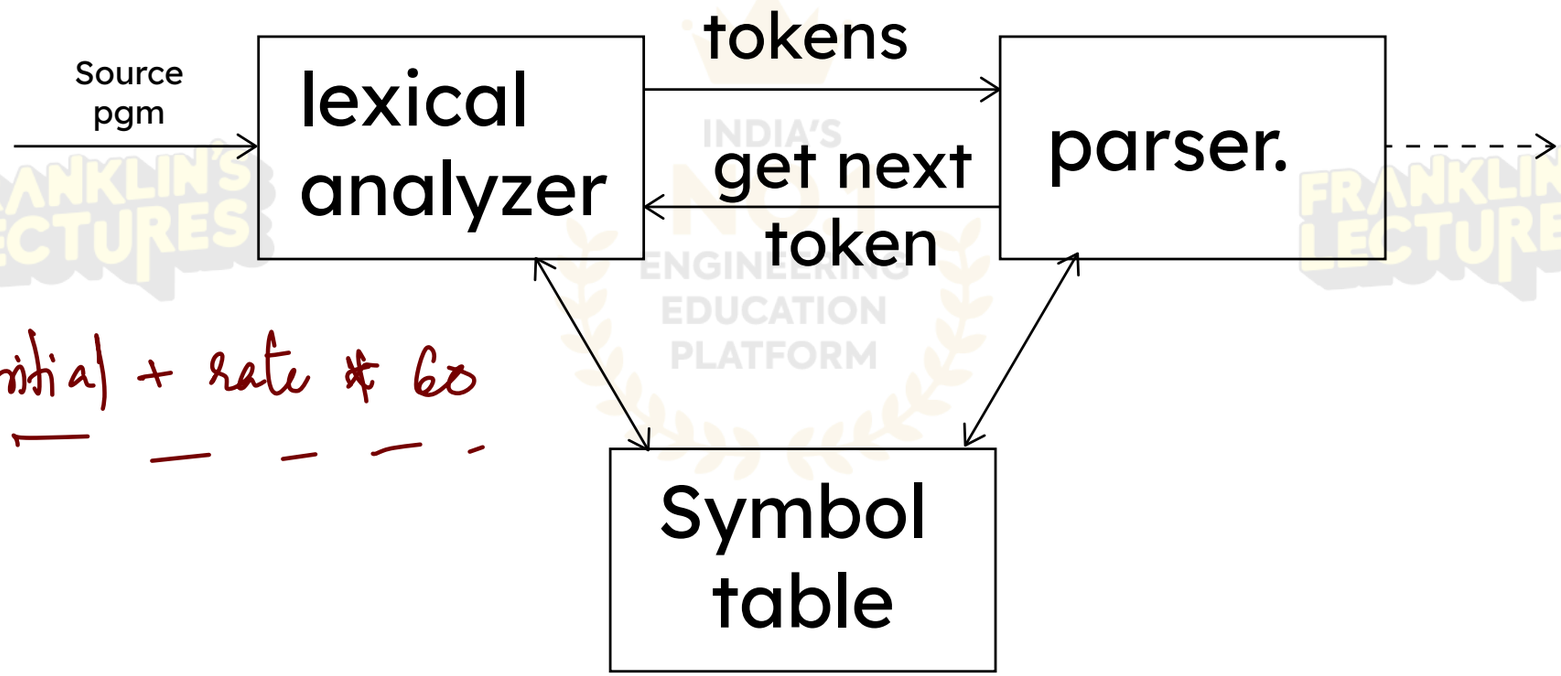
When the source language is also the implementation language & the source text to be compiled is actually a new version of the compiler itself," the process is called bootstrapping.

→ Bootstrapping is the technique for producing a self compiling compiler that is written in the source programming language that it intends to compile.

LEXICAL ANALYSIS

Source pgm → [LA] → seq. of tokens

- **Role of Lexical Analyzer**



position = initial + rate * 60

- Lexical analyzer is the first phase of the compiler. Its main task is to read the input characters and produce as a sequence of tokens as output that parser uses for syntax analysis. This interaction is represented in the above figure
- Upon receiving "get next token" message from the parser the lexical analyzer reads & input characters until it identify next token. Other functions of lexical analyzer are

1) Eliminate comments and white spaces (in the forms of blanks, tabs and new line characters)

2) Correlating error messages from the compiler with the source programs. For eg:- The lexical analyzer may keep track no. of new line characters seen. so that a line number can be associated with an error message

- In some compilers, the lexical analyzer is in charge of making the copy of source pgm with the error messages marked in it.

line1 int a ;

line2 int R

line3 a+b = c

err

line2 :

Tokens, Patterns and lexemes

- Tokens: We treat token as terminal Symbol in the grammar for the source language using bold phase names."
- eg:- **id**, **const**
- lexemes
- A lexeme is a sequence of the characters in the source program that is matched by a pattern for a token.
- eg:- in the pascal statement, const pi=3. the substring Pi is a lexeme for the token identifier

Pattern

- A pattern is a rule describing a set of lexemes that can represent a Particular token in the source pgm. eg:- The pattern for the **token const** is just the single const that spells the Key word
- In most programming languages the following constructs are treated as token. Keywords, operators, identifiers, constants, literal strings & punctuation symbols such as parenthesis, commas and semicolons.

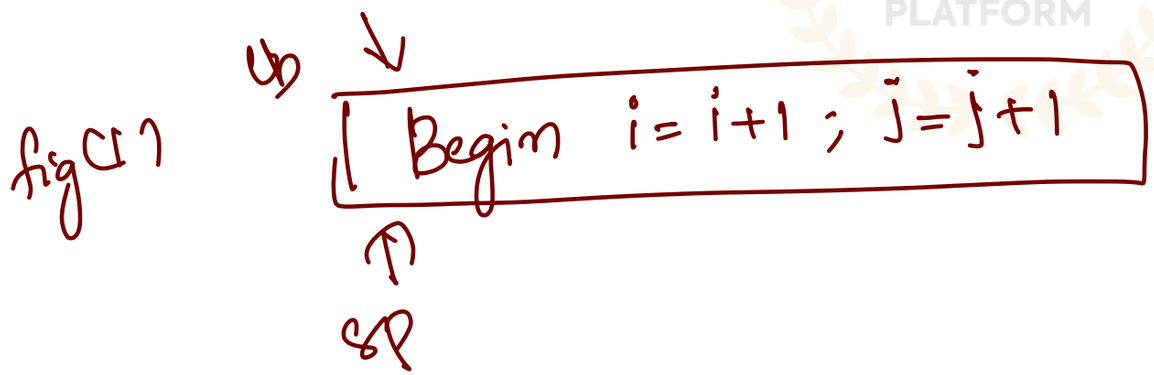
Examples of tokens

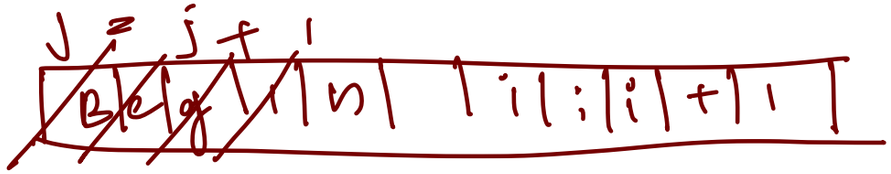
Tokens	sample lexemes	Information description of pattern
const	const	const
if	if	if
relation	<,>,=,<=,>=,<>	< or > or = or <= or >= or <>
id	Pi, count, D2.	letter followed by letterse digits
num	3.14,6.02	any numberic constant
literal	"good"	any character blw “&”

Input Buffering

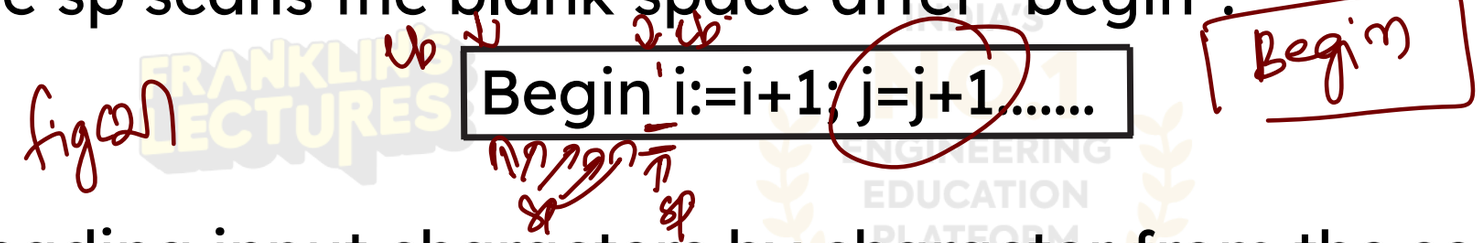
- The lexical analyzer uses two Pointers to read tokens. They are 'lb' (lexeme beginning) pointer that indicates the beginning of the lexeme and "sp" (search-pointer) that keeps track of the portion of the input String scanned.
- lb sp `Begin i= i+1; j = j+1`

fig (1): Initial position of the pointers"lb" and "sp"





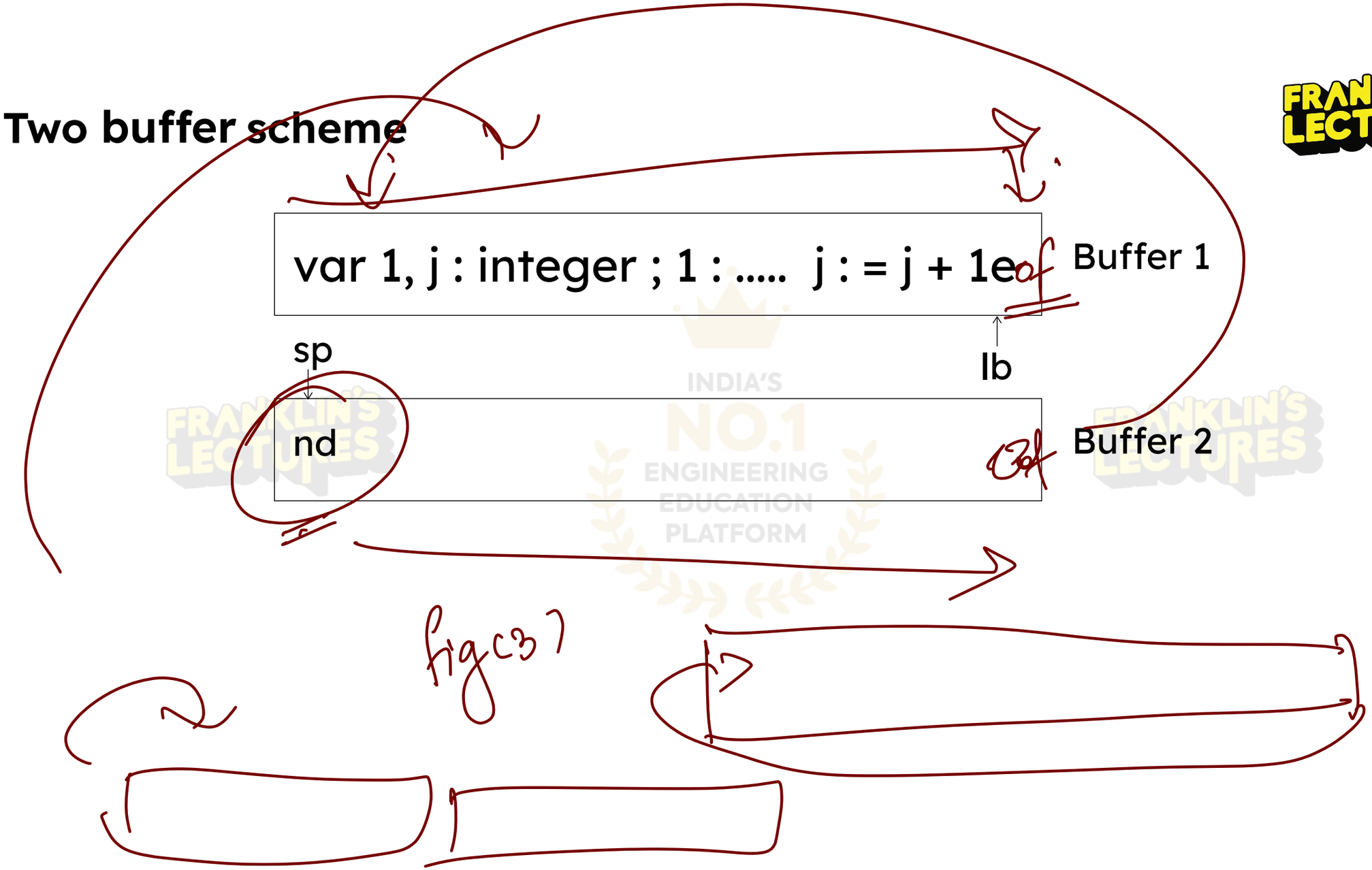
- Initially both pointers point to the beginning of a lexeme . The search ptr then starts scanning f/w to search for the end of the lexeme. The end of the lexeme, in this case is indicated by the blank space after "begin . The lexeme is identified only when the sp scans the blank space after "begin".



- Reading input characters by character from the secondary storage is costly. A block of data is read first into a buffer, and then scanned by the lexical analyzer . For this purpose buffering methods are used. Commonly used buffering methods are.

- 1) One buffer Scheme There are problems if a lexeme crosses the buffer boundary. To scan the rest of the lexeme, the buffer has to be refilled thereby overwriting the first part of the lexeme.
- 2) Two buffers Scheme: Here, buffers 1 and 2 (Fig 3) are scanned alternatively When the end of the current buffer is reached, the other buffer is filled. Hence the problem encountered in the one buffer method is solved
- In this scheme, the second buffer is loaded when the first buffer becomes full. Similarly the first buffer is filled when the end of the second buffer is reached. Then the "sp" pointer is incremented.

Two buffer scheme



- Hence two tests have to be done to increment the "sp" pointer. This can be reduced to one test if we include a sentinel character (a special character, not a part of the input programs) at the end of the buffer Examples of these characters EOF. So only if the EOF character encountered Second check is made as to which buffer has to be refilled and the action is performed.
- **Input buffering**

  Storing a block of input data in buffer to avoid costly access to secondary storage each time.

- **Specification of tokens**

Scanners are special pattern matching processors For representing Pattern of Strings of characters, Regular expressions (RE) are used." The term alphabet or character class denotes any finite set of symbols.

Typical examples of symbols are Letters and characters. The set {0, 1} is the binary alphabet. set {0, 1 ... 9} digit alphabet

A String over some alphabet is finite sequence of symbols drawn from that alphabet.

The term language denotes any set of strings over some fixed alphabet

For eg:- regular expression for the token identifier is,

letter (letter/digit)*.

Steps to specify tokens.

1. identify token categories
   - keyword - if, else, while, ...
   - constants, variable fn name - tmp, mm
   - operators =, -, + ...
   - literals : 123, 3.14, 'a', "hello"
2. Define RE for each token type
3. Convert RE to Finite Automata → RE → NFA → DFA
4. generate token stream.

identifier $[a\text{-}zA\text{-}Z][a\text{-}zA\text{-}Z0\text{-}9]^*$

integer literals $[0\text{-}9]+$

lexer reads source code & matches patterns using DFA to generate tokens.

whitespace $[\backslash t \backslash n \backslash r]+ \Longrightarrow$ ignored.

- **Recognition of Tokens**

  The tokens obtained during lexical analysis are recognised using a finite automaton.

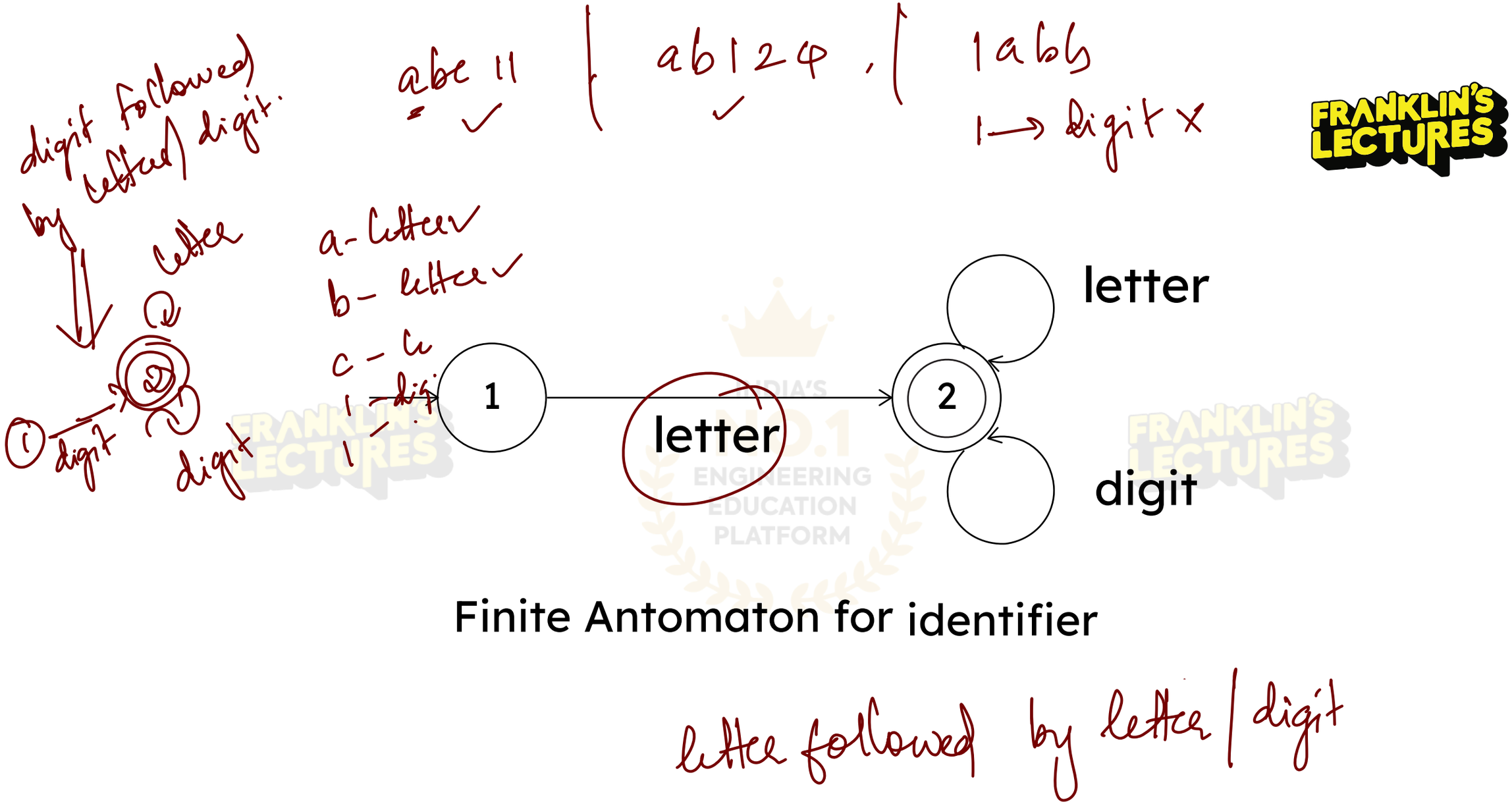
  Finite Automata:- FA can be used to describe the process of recognizing patterns in input strings and so they can be used to construct scanners. FA or finite state machines are a mathematical way of describing particular kinds of algorithms It produces the transition diagram for regular expression. It takes as input a Particular string and verifies whether the string belongs to the language or not. A regular definition for an identifier is given below:

letter ⟶ a/b/c..../z or letter ⟶ [a-z]

digit 0/11/2/...9 or digit ⟶ [0-9].

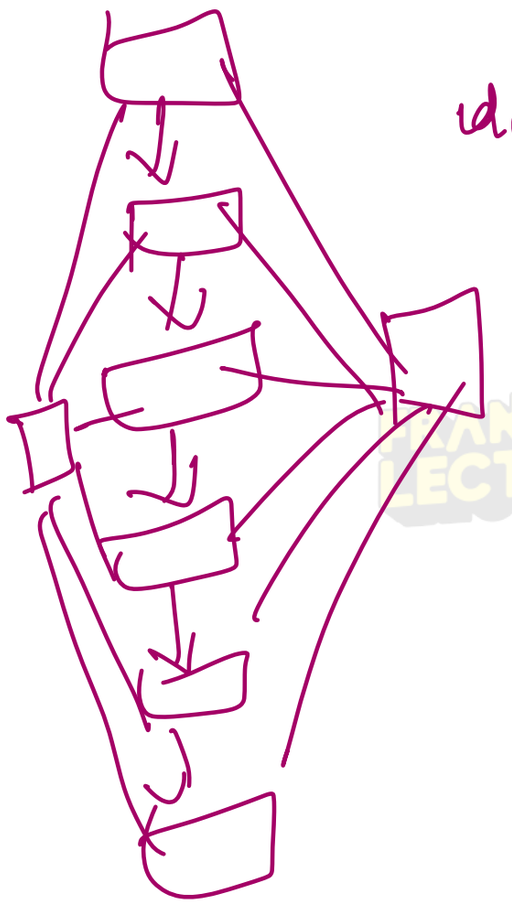
identifier ⟶ letter (letter/digit)*.

Thus an identifier consists of a letter followed by any number of letters or digits. The finite automaton for the identifier can thus be represented as shown in the following figure.



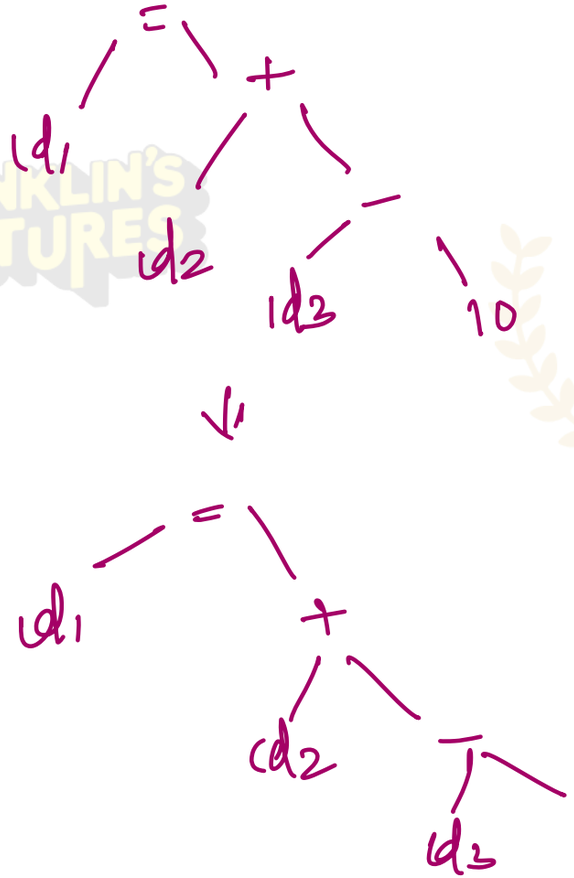
Finite Antomaton for identifier

phases of compiler



$a = b + (c - 10)$

$id_1 = id_2 + (id_3 - 10)$



a, b, c — Floating point

temp1 = inttoreal(10)
temp2 = id3 − temp1
temp3 = id2 + temp2
id1 = temp3

temp1 = id3 − 10.0
id1 = id2 + temp1

```
MOVF  id3, R2
SUBF  #10.0, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

set of characters → tokens → parse tree (grammatic phase) → type checking → intermediate code generate → code optimization → code generate

→ 3 address code

Issues in L.A / reason why Lexical Analysis is seperated from Syntax Analysis.

→ Simplicity of Design
→ Efficiency
→ portability.

FRANKLIN'S LECTURES

THANK YOU