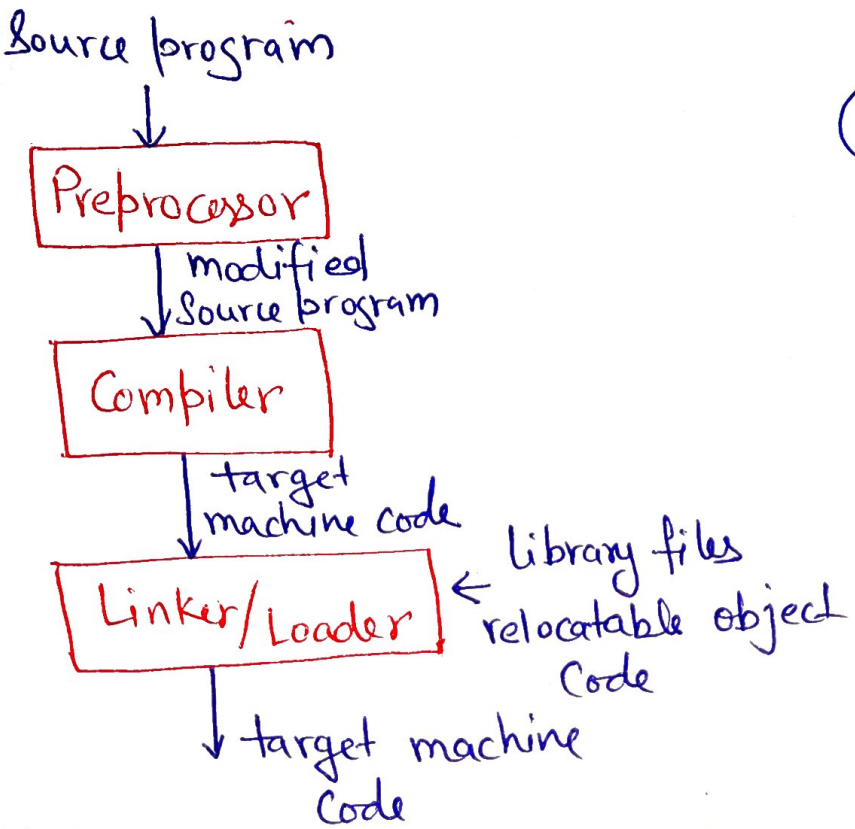
LANGUAGE PROCESSING SYSTEM :-

COMPILER DESIGN

UNIT - 1

(Pragya Gaur)



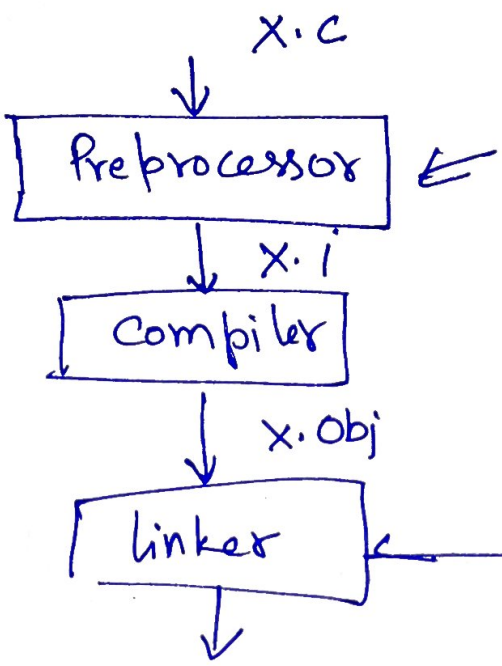
In addition to a compiler several other programs may be required to create and execute target program.

* A source program may be divided into modules stored in separate files.

- The process of collecting the source program is done by a separate program preprocessor.
  The modified source program is then fed to the compiler.
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.
- The linker resolve external memory addresses, where the code in one file may refer to a location in another file.
- The loader then put together all the executable object files into memory for execution.

Pragya Gaur

Steps in C-Compilation-



(include)

stdio.h
(where printf is declared)

lib.a
(lib)
where printf is defined

1) #define — macro

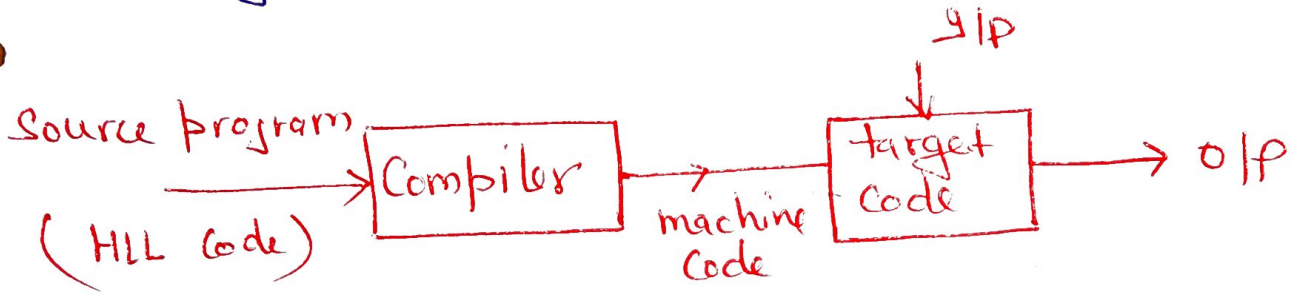
2) file inclusion - #include

Translator :-

Translator is a program that read a program in one language (source code) and translate it into a program of other language (target code)

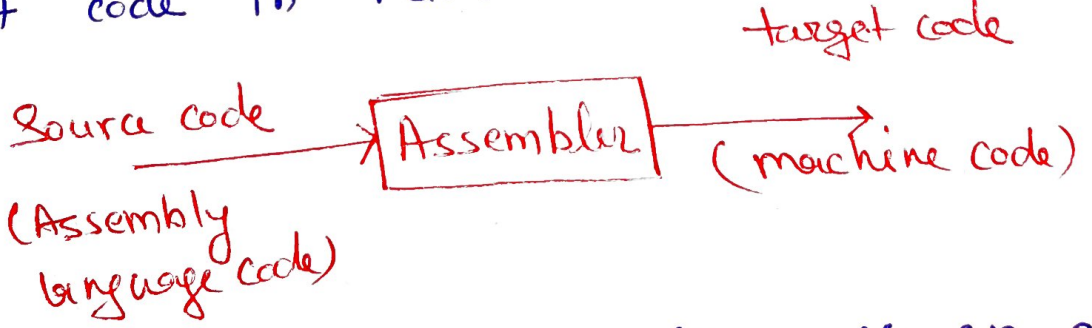
Compiler :-

A compiler is a translater that takes the I/P program written in high level language & produce the target program in machine code.



Assembler :-

An Assembler is a translator that translate the source code written in assembly code into the target code in machine code.



In other words we can say ~~ass~~ an assembler is translator for assembly language.

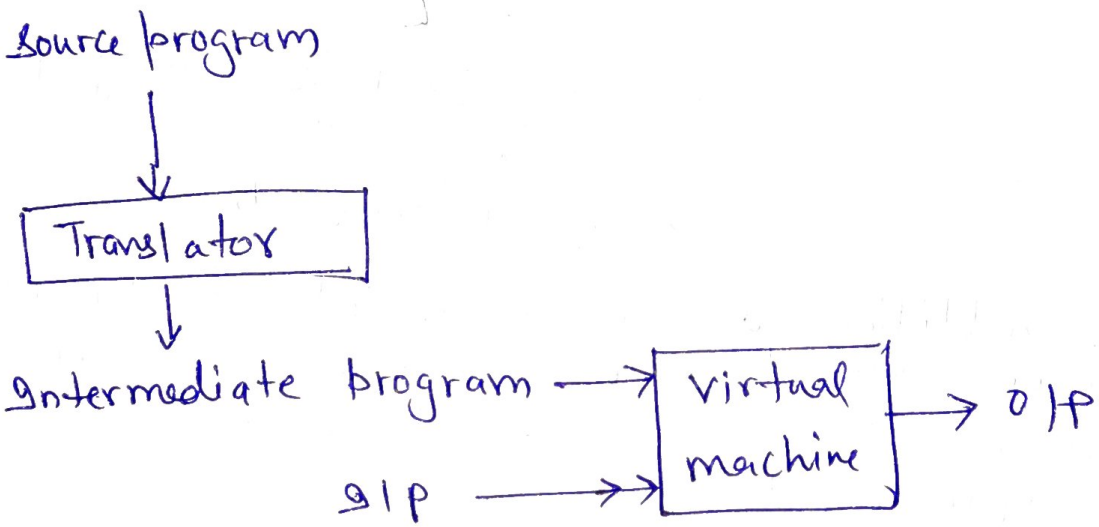
→ hybrid Compiler:-

Java language processors combine compilation and interpretation,

A java source program may first be compiled into an intermediate form called 'bytecodes'.

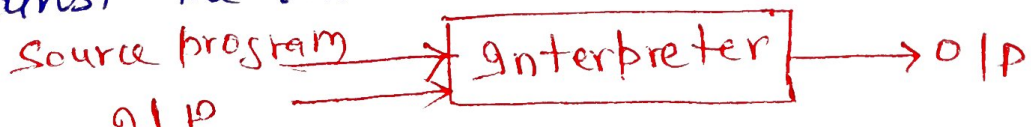
The bytecodes are then interpreted by a virtual machine.

* to achieve faster processing of I/P to O/Ps, some java compilers (just in time compilers) translate the bytecodes into machine language immediatly before they run the inter mediate prog. to process the I/P



→ Intrepreter:-

An intrapreter reads a source program written in HLL as well as data for this program and runs the program against the data to produce same result.



* Interpreter is not a translator.

Translater produces the object code, but interpreter runs the program and produce the result

Prajya Gaur

Comparision between Compilation and Interpretation :-

Compiled languages have a bias towords static properties. (all compiling decisions are made using source text at translation time) where as Interpreted language deals with dynamic properties.

"static property" of a program is a property that is evident from the program text.

"dynamic property" is evident only upon running the program.

The comparision of compilation & Interpretation can be list out as-

1). Compilition can be more efficient then Interpretation
2) Interpretation can be more flexibile then compilation

1☆. Compiler translate the source program once for all while an interpreter examines the program repeatedly This repeatation (repeated examination) is a source of inefficiency.

2☆. but this repeated examination allow the interpretation to be more flexible then compilation, As an interpretar directly runs the source program, so

Structure of Compiler :-

the structure of compiler can be partitioned into two parts -

1) Analysis — front end
2) Synthesis — back end

Analysis :-

The analysis part breaks up source program into constituent pieces and impose a grammatical structure on them, and then by this structure create an intermediate representation of the source program

Analysis part also collect the information about the source program and store it in a data structure (symbol table)

Synthesis part :-

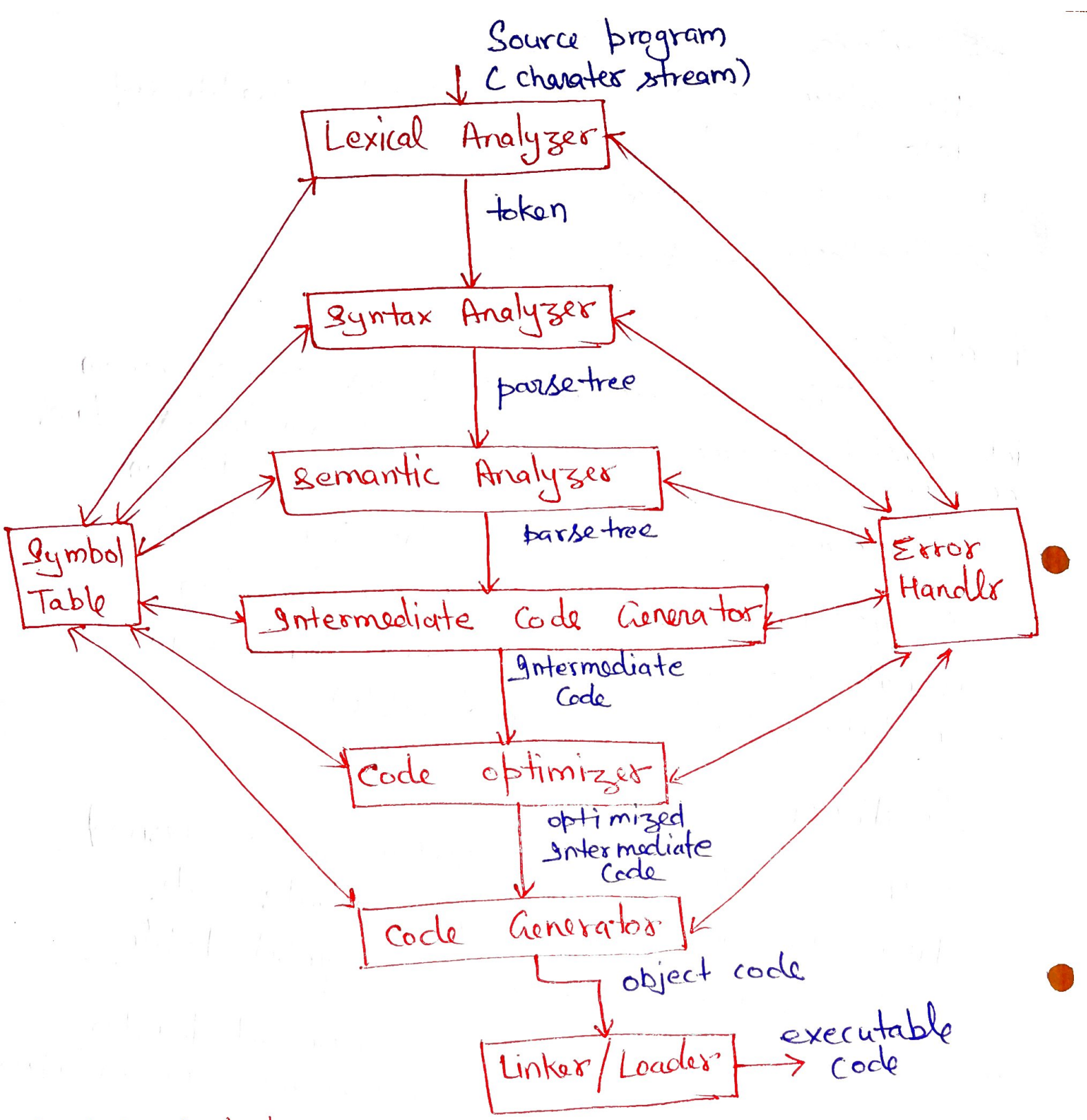
The synthesis part constructs the desired target program from the intermediate code and the information in the symbol table.

The structure of compiler can be represented into several phases, each phase transforms one representation of source program to another -

phases are -

1) Lexical Analysis
2) Syntax Analysis
3) Semantic Analysis
4) Intermediate Code Generation
5) Code optimization
6) Code Generation
7) Symbol table
8) Error handling

Progra Gau



Lexical Analysis :-

Separates the character of the source program into groups that logically belong together.

These groups are called tokens, and the o/p of lexical Analyzer is a stream of tokens.

example- Keyword - Do, If, WhILE
identifier - x, NUM, Abc
operater - +, -, /, *, <, > ---
punctuation symbol - (,); , , . ---

Lexical Analyzer reads the source program character by character and return the tokens.

Token-

A token describes a pattern of characters having same meaning in the source program. It can be described as a pair consisting of token name an optional attribute value.

lexeme -

lexeme is a sequence of character in the source program that matches the patterns for a token and identify by lexical Analyzer as instance of token.

example-

token	lexeme
if	if
id	bi, score, position
comparision	<= >=
number	1, 2, 3, 4...
literal	"Compiler Design"

ex-

position = initial + rate * 60

tokens returned by lexical analyzer are -

<id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

Syntax Analyzer-

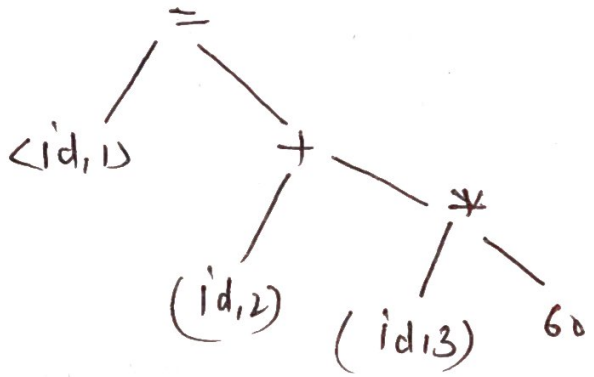
Syntax analyzer groups tokens into syntactic structure.

The intermediate code generator uses the structure produced by syntax analysis to create a stream of simple instructions.

The parser uses the tokens produced by lexical analyzer to create a tree-like structure "Syntax tree" to present the grammetical structure of the token stream.

example - id = id + id * 60

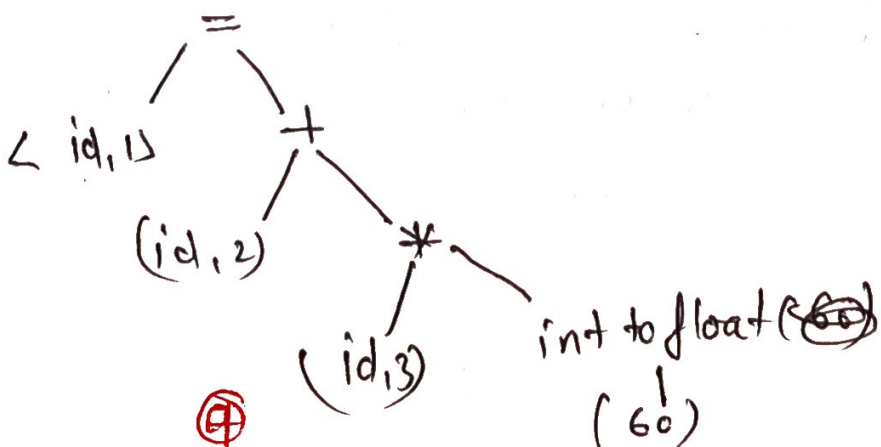
token - <id> <=> <id,2><+><id,3><*><60>



Semantic Analysis →

Semantic analyzer takes the syntax tree and the info^n in the symbol table to check the source program for semantic consistency with the language definition.

ex.



Intermediate Code Generation -

The intermediate code generater uses the structure produced by semantic Analysis to create a stream of simple instructions.

ex -.

$t_1 = \text{inttofloat}(60)$

$t_2 = id3 * t_1$

$t_3 = id2 + t_2$

$id_1 = t_3$

Code optimization :-

Code optimization is an optional phase designed to improve the intermediate code so that the ultimate object program run faster and / or takes less ~~time~~ space.

The output of code optimization is another Intermediate code.

ex -.

$t_1 = id3 * 60$

$id1 = id2 + t_1$

Code Generation -

It produces the object code by deciding the memory location for data, selecting the register in which each computation is to be done.

ex -

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id1
ADDF  R1, R1, R2
STF   id1, R.
```


Pragya Gaur

(Symbol table) Table management :

bookkeeping portion of the compiler keeps track of the names used by the program and record essential info^n about each (tyke: int)

The data structure used for this is symbol table.

Ex- find the o/p generated by compiler after each phase -

1) a = b+c+d*2.0

2) a = b+c*20

3) X = Y-Z

4) expression = Expression + term * factor

Error handler →

Whenever ~~an~~ a phase recognize an error Error handler has been invoked,

many techniques are used for error handling

1) panic mode error recovery
2) phase level error recovery

the error recovery routine either opt an recovery machanism or display the error message on the screen.

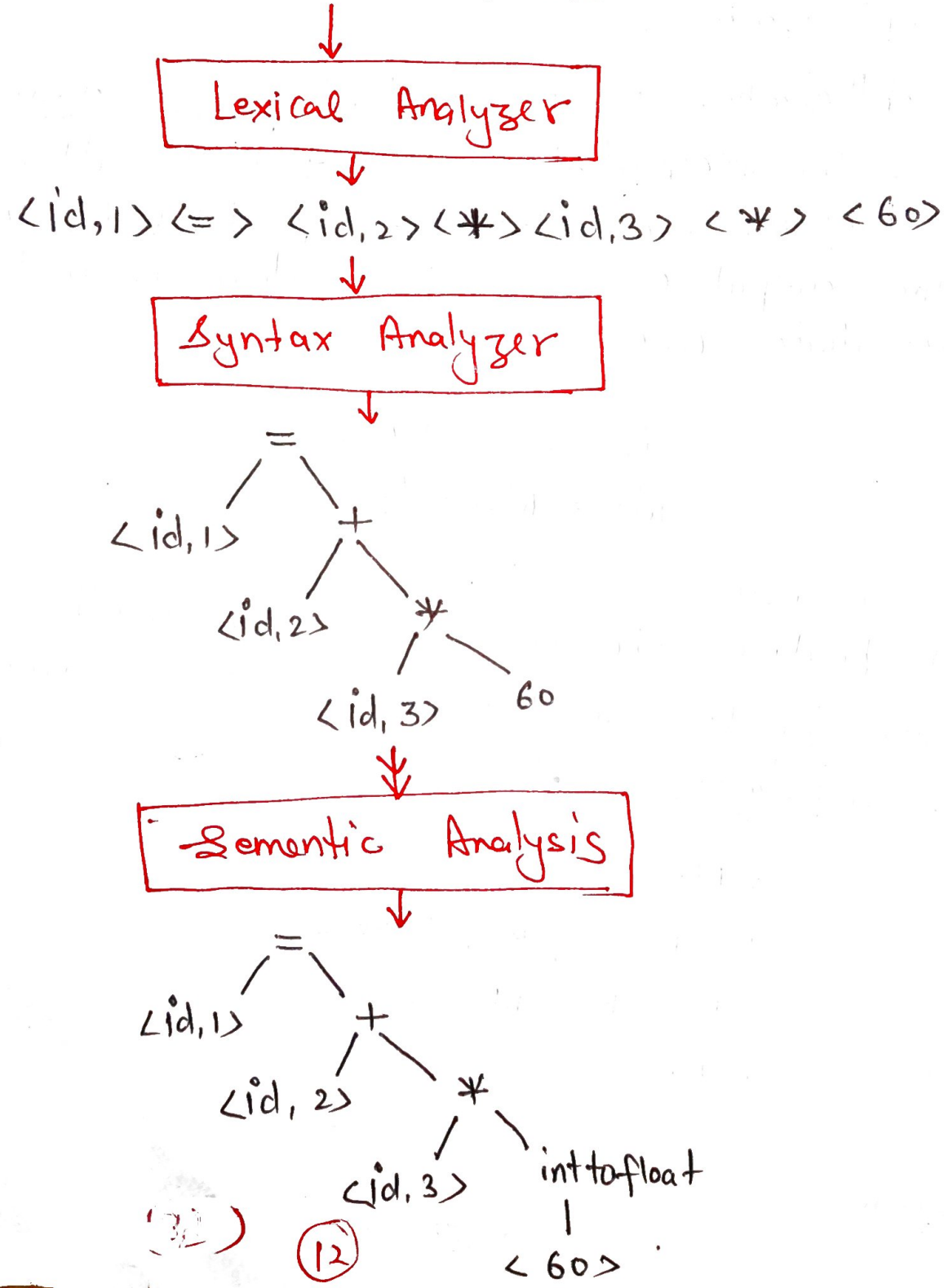
Translation of an assignment statment

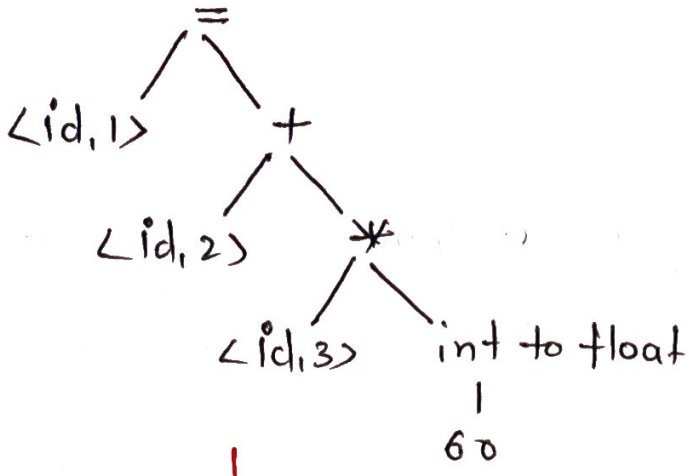
Consider the assignment statment —

position = initial + rate * 60

here position, initial, rate → identifier
+, *, = → symbols for addition, multiplication & assignment operator

⇒ position = initial + rate * 60





↓

Intermediate Code Generator

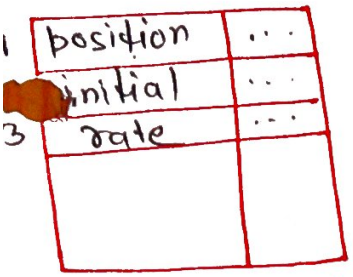
↓

$t_1 = \text{inttofloat}(60)$

$t_2 = id3 * t_1$

$t_3 = id2 + t_2$

$id1 = t_3$



Symbol table

↓

Code optimization

↓

$t_1 = id3 * 60.0$

$id1 = id2 + t_1$

↓

Code Generator

↓

(load float) LDF R2, id3

(multiply float) MULF R2, R2, #60.0

LDF R1, id2

(Add float) ADDF R1, R1, R2

(Store float) STF id1, R1

Phases –

Phases of a compiler are the sub-tasks that must be performed to complete process

Pass–

Passes refer to the no. of times the compiler has to traverse through the entire program.

Cross Compiler :-

A cross compiler is a compiler that may run on one machine and produce object code for another machine,

In other words language of compiler is different from the object code language.

ex.

C_A^{SB} → is a compiler written in language A, taking input in language S and producing object code in language B.

Bootstrapping:

Three language are associated with a compiler

C_S^{AB}

S - Language in which compiler is written
A - Source language
B - target language

In general -

The language of compiler and the target language are usually the same language on which compiler is working.

If the compiler is written in it's own language then the problem arises "How to compile the first compiler".

The process of making a language available on a machine is called bootstrapping.

Bootstrapping language S on machine A:

produce a compiler written in language of machine A taking the source code in language S and produce object code in language A

C_A^{SA}

Pragya Gaur

for doing this first we take a language R, which is a small part of language S. (A subset of language S)

Write a compiler of R in language A → C_A^{RA}

then construct a compiler written in R, taking the source code in S and producing object code in A.

Compiling this compiler C_R^{SA} on compiler of R make a full length compiler of S.

Bootstrapping a compiler for a second machine B with the help of compiler for A:-

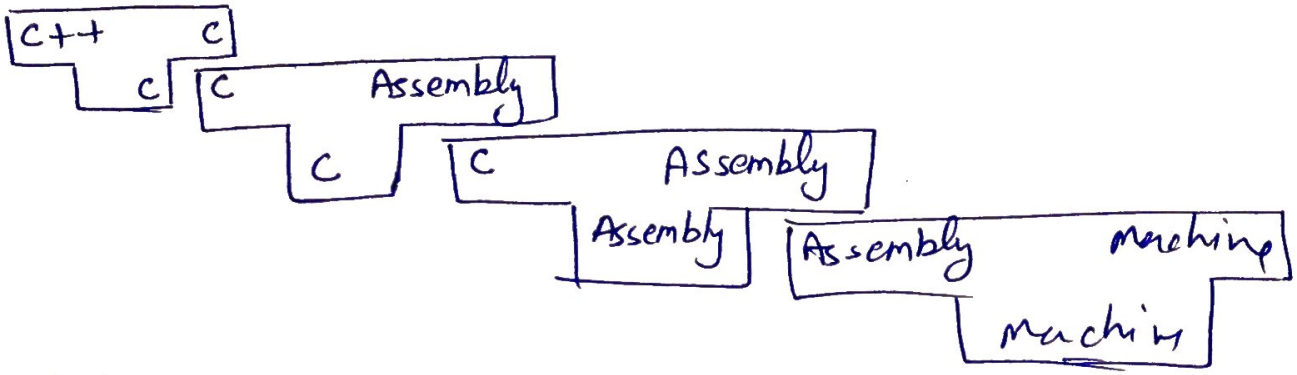
wanna construct C_B^{SB} with the use of C_A^{SA}

first construct a small compiler in S → C_S^{SB}

then compile this on C_A^{SA}

It will give a cross compiler C_A^{SB} and again compile C_S^{SB} on C_A^{SB} will produce C_B^{SB}, a full length compiler for language B.

ex-1. to make c++ available on machine A



Step-

(1) Create an assembler using machine code
(2) Create a simple c compiler using assembler
(3) Create a better c compiler using C
(4) Create a simple c++ compiler using C

Example-2

Suppose C_A^{CA} is available (A indiate a window based machine)
and our motive is to ~~b~~ ~~prcap~~ make C available on machine B (for example unix based system)

$$C_C^{CB} \rightarrow C_A^{CA} \rightarrow C_A^{CB}$$ (cross compiler

$$C_C^{CB} \rightarrow C_A^{CB} \rightarrow C_B^{CB}$$

Write a ~~small~~ compiler in C language which produce target code in B's language, and compile this program on machine A's compiler, this will produce a cross compile C_A^{CB} (written in A and producing target code in B)
Compiler C_C^{CB} on C_A^{CB} will produce a full length compiler for machine B.

✱ Compiler Construction tools :-

The compiler writer uses modern s/w development environment containing tools (ex- language editors, version managers, debuggers etc), in addition to these tools, other more specilized tools have been created to help implementation of various phases of a compiler.

These tools uses specilized language for specifying and implementing specific components.

The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

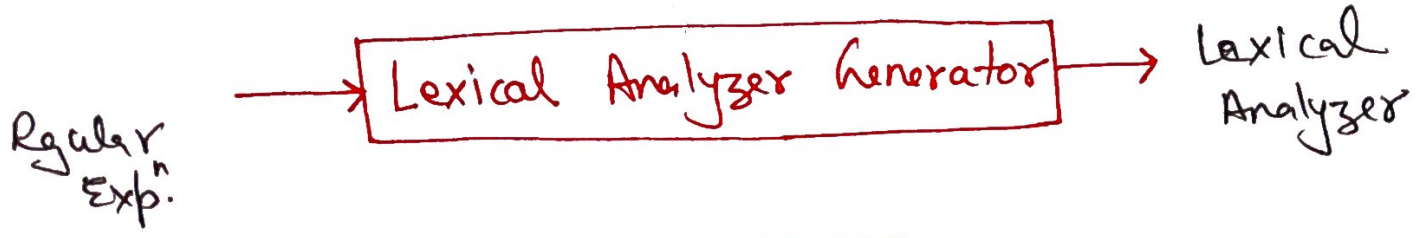
Some commonly used compiler construction tools are -

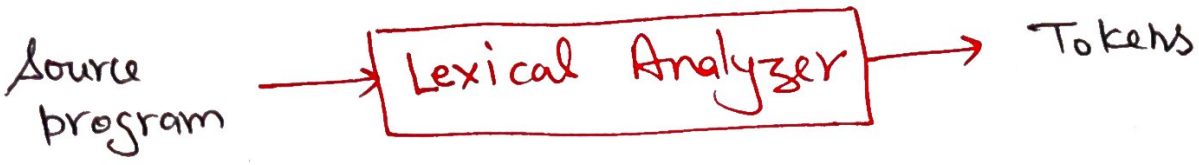
1. parser generators.
2. Scanner generators.
3. Syntax directed translation engines.
4. Code Generator generators.
5. Data flow analysis engines.
6. Compiler Construction toolkits.

✱ parser generator automatically produce syntax analyzer from a grammatical description of a programming language.

* Scanner generator produce lexical analyzers from a regular-expression description of the tokens of a language.
* Syntax directed translation engines produce collection of routines for walking a parse tree and generating intermediate code.
* Code generators generators produce a code generator from a collection of rules for translating each operations of the intermediate language into the machine language for a target machine.
* Data flow analysis engines facilitate the gathering of info about how values are transmitted from one part of a program to each other part.
* compiler construction toolkit provides an integrated set of routines for constructing various phases of a compiler.

~~LEX~~ (The Lexical Analyzer Generator):



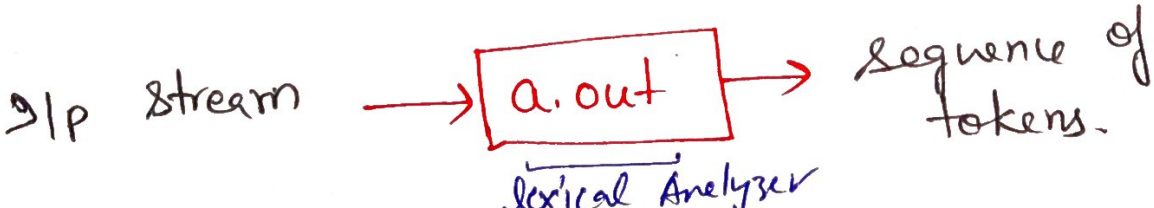


LEX is an exampler of lexical Analyzer Generator.

LEX →

The input notation for the lex tool is referred to as the lex language and the tool itself is lex compiler.

Lex compiler transform the i/p pattern into a transition diagram & generate code in a file Lex.yy.c that simulate this transition diagram.



here a.out is a working compiler that takes the i/p stream as i/p and produce the sequence of tokens.

Structure of LEX:-

The format of Lex program is

Declarations
%%
translation rule
%%
auxiliary functions.

translation rule have the form

pattern {Action}

Lexical Analyzer :-

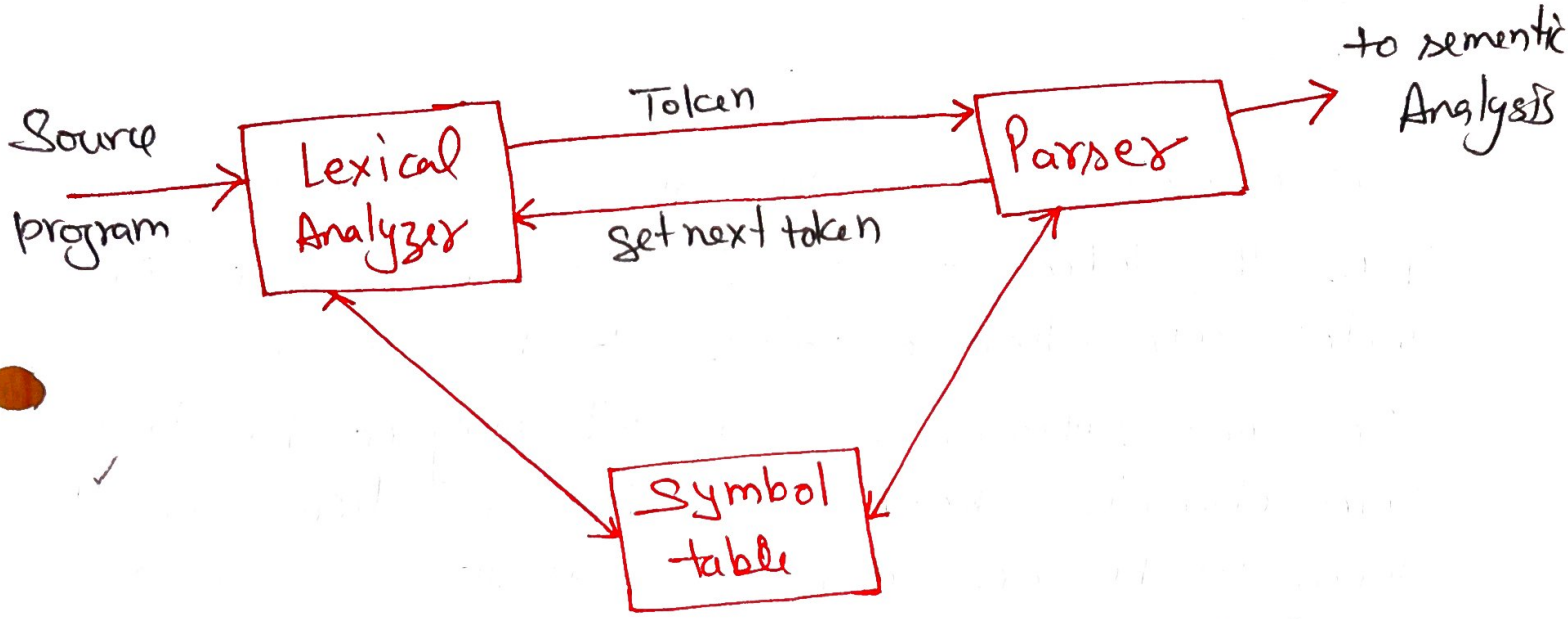
In the first phase of compiler, the main task of lexical Analyzer is to read the ip character of source program, group them into lexemes ~~in the source program.~~

And produce a sequence of tokens for each lexeme in the source program.

This stream of tokens is sent to the parser for syntax analysis.

When the lexical Analyzer discovers a lexeme constituting an identifier, it need to enter that lexeme into the symbol table.

In some cases, info" regarding the the kind of Identifier may be read from the symbol table by the lexical analyzer to assist it in determining the proper token it must pass to the parser.



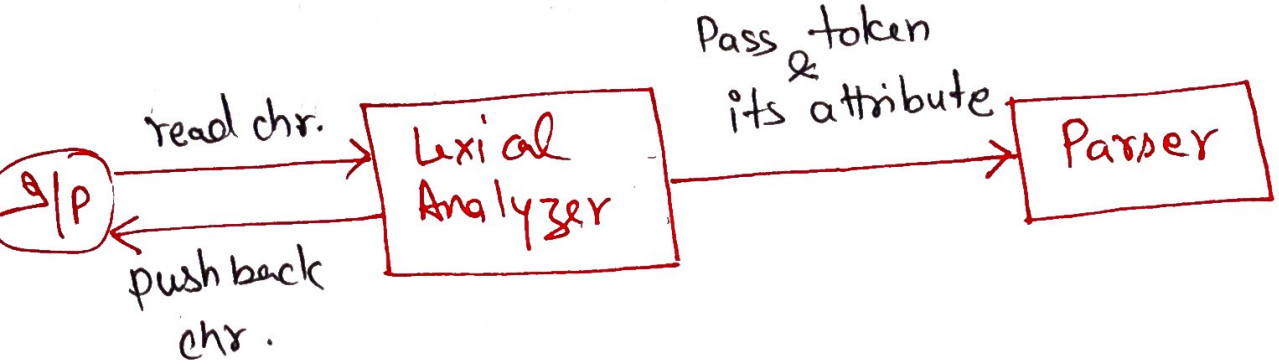
lexical Analyzer also perform certain other task besides identification of lexeme.

1) Stripping out commants and white space.
2) Correlate error message generated by compiler to the source program.

lexical Analysis Vs parsing:

1) Simplicity of Design
2) improve efficiency
3) Compiler portability is enhanced

Input Buffering :-



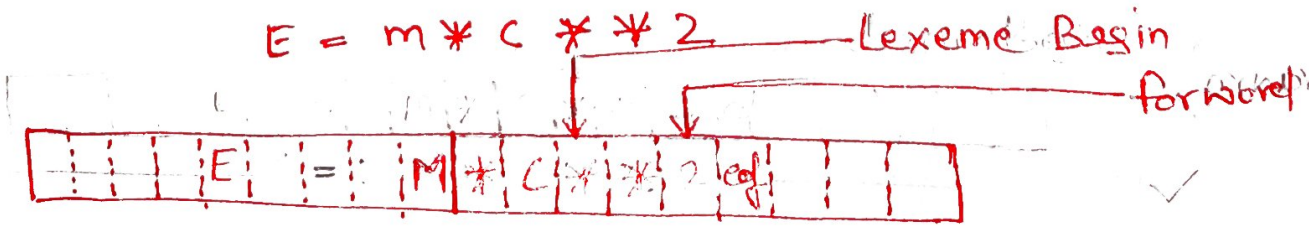
read character from the I/p, groups them into lexemes, pass the tokens formed by the lexems together with their attribute values to the parser.

In some situation, the lexical analyzer has to read some character ahead before it can decide on the token to be forworded to the parser.

then extra character has to be pushed back onto the I/p because it can be the begining of the next lexeme.

The implementation of reading & pushing back the character is usually done by setting up an I/p buffer.

example -



because of the amount of time required to process character and the large no. of characters processed during the compiliation of a large source program specilized buffering techniques are used to

reduce the overhead required to process a single i/p character.

We use a buffer divided into 2 N-character halves. each buffer is of the same size N, and N is usually the size of a disk block.

We read N i/p characters into each half of the buffer with one system read command, rather than invoking a read command for each i/p character.

If fewer than N-character remains in the i/p, then a special character 'eof' is read into the buffer after the i/p characters.

'eof' marks the end of source file & it is different from any i/p character.

for input buffering two pointers are maintained

1) lexeme begin
2) forword

lexeme begin marks the begining of the current lexeme, forword scans ahead until a pattern match is found.

The string of charater b/w these two pointers is the current lexeme.

Prayya Gaur

Initialy both pointer points to the first character of the next lexeme to be found.

forword pointer scan ahead until a match for a pattern is found.

once the lexeme is determined, the forword pointer is set to the character at it's right end. After the lexeme is processed, both pointers are set to the character immediatly after lexeme.

Simple Approch to design of lexical Analyzer-

we may use flowchart to describe the behaviour of any program.

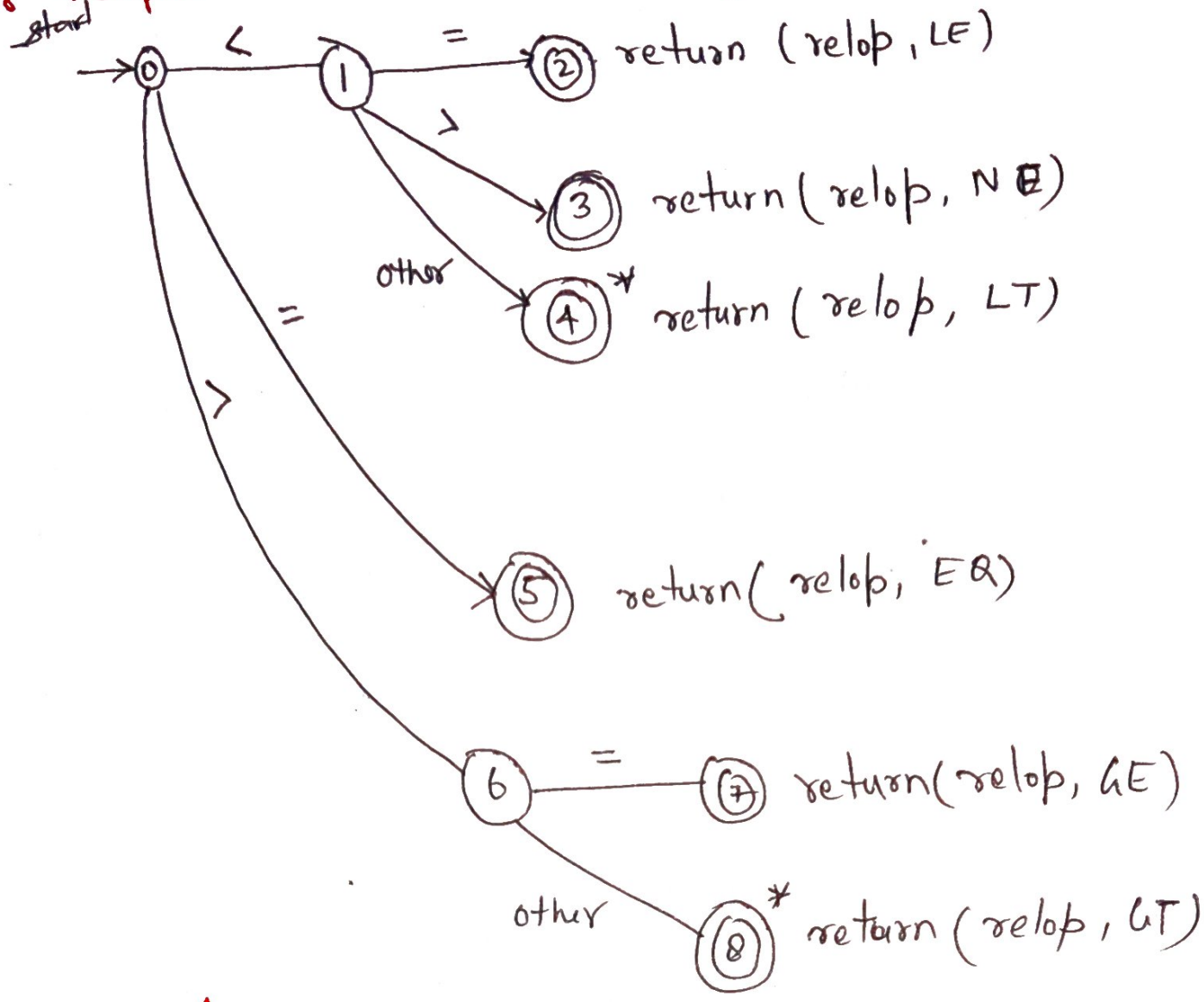
A specilized kind of flowchart 'Translation diagram' is useed

In transition diagram, the boxes of the flow chart are drawn as circles and called states.

the states are connected by arrows called edges.

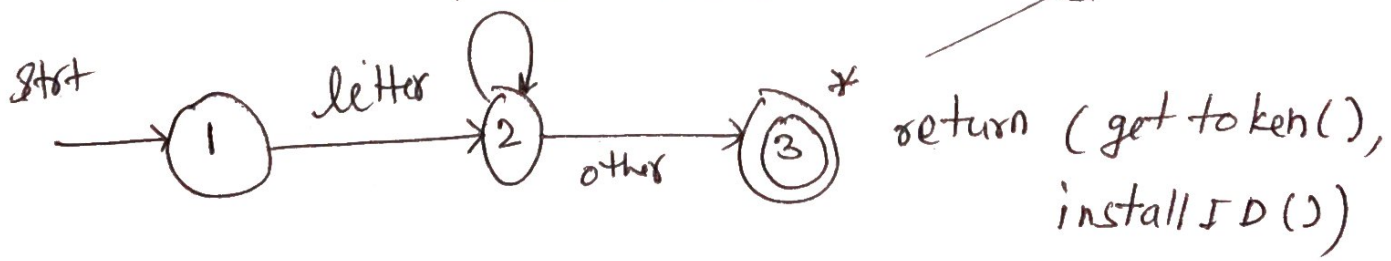
Transition Diagram for tokens

for relop –



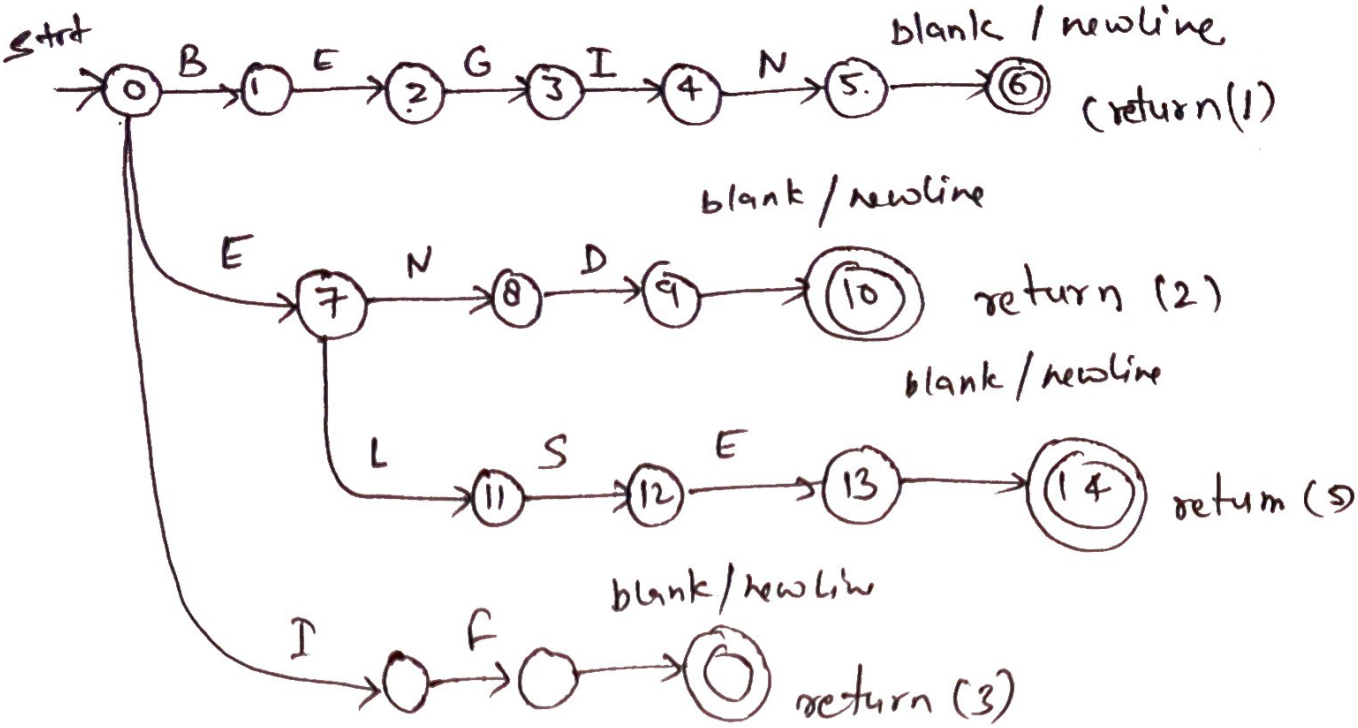
for identifier

TD for reserved word & Identifier –

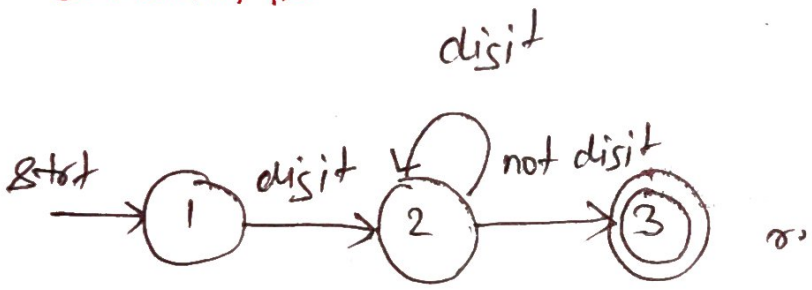


* means we must retract the I/p one position.

Transition Diagrams –
for keyword –



Constant :–



Language :-
Terminology ⇒

Alphabet :-
A finite set of symbols.

String :
finite sequence of symbols on an alphabet.
ϵ is the empty string,
$|s|$ denote the length of string.

Language -
Language is a set of strings over some fixed alphabet.

ex-
$\{\epsilon\}$ the set containing empty string is a language.
the set of all possible identifiers is a language.

operations on language :-

1) Concatenation
$$L_1 L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$$

2) Union
$$L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$$

3 Exponentiation
$$L^0 = \{\epsilon\}, \quad L^1 = L, \quad L^2 = LL$$

1, Kleene Closure L^* (all set of string including ϵ)

0. Positive closure L^+ (all set of string except ϵ }

Regular Expressions –

to describe tokens of a programming language we use regular exp^n.

each regular expression denote a langugr.

A language denoted by a regular expression is called as a regular set.

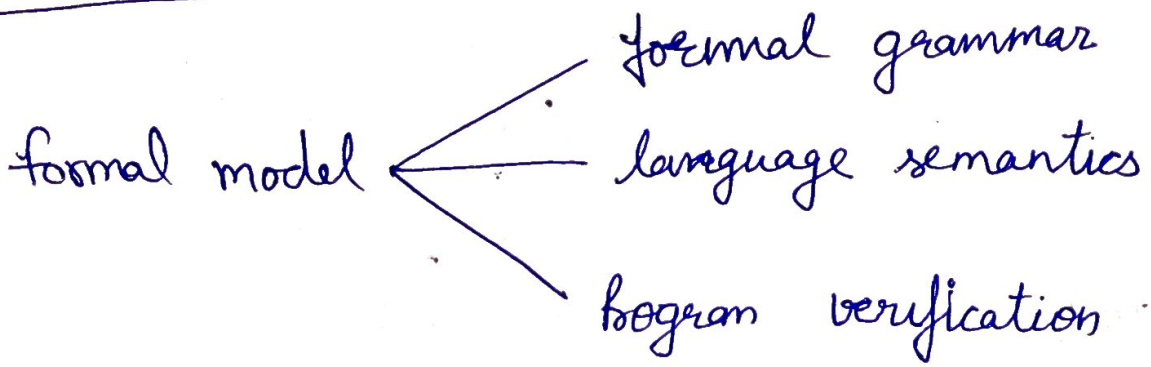
ex –

Regular exp^n over alphabet Σ

Reg. exp^n	Language
ϵ	$\{\epsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)(r_2)$	$L(r_1)\, L(r_2)$
$(r)^*$	$(L(r))^*$

* A regular string is a special text string for ~~Thats~~ describing a search pattern.

Modeling language properties -



formal grammar → BNF and regular grammar

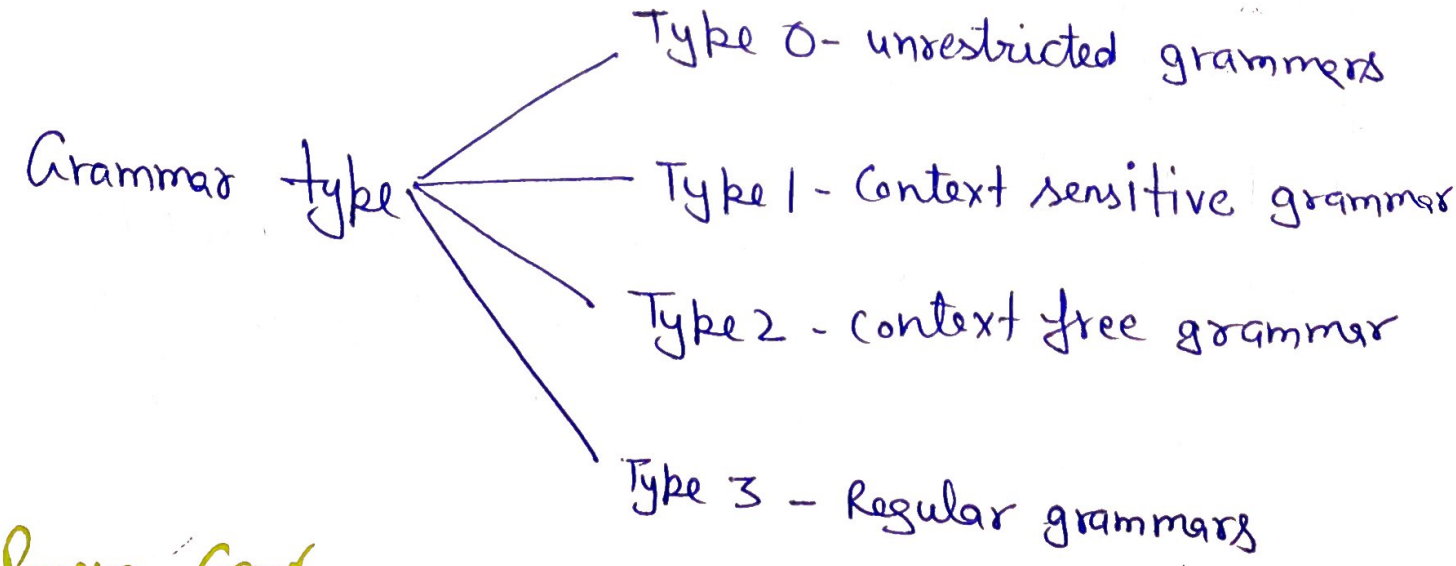
language semantics → This model appends semantics information to the BNF description of a language

Program Verification -

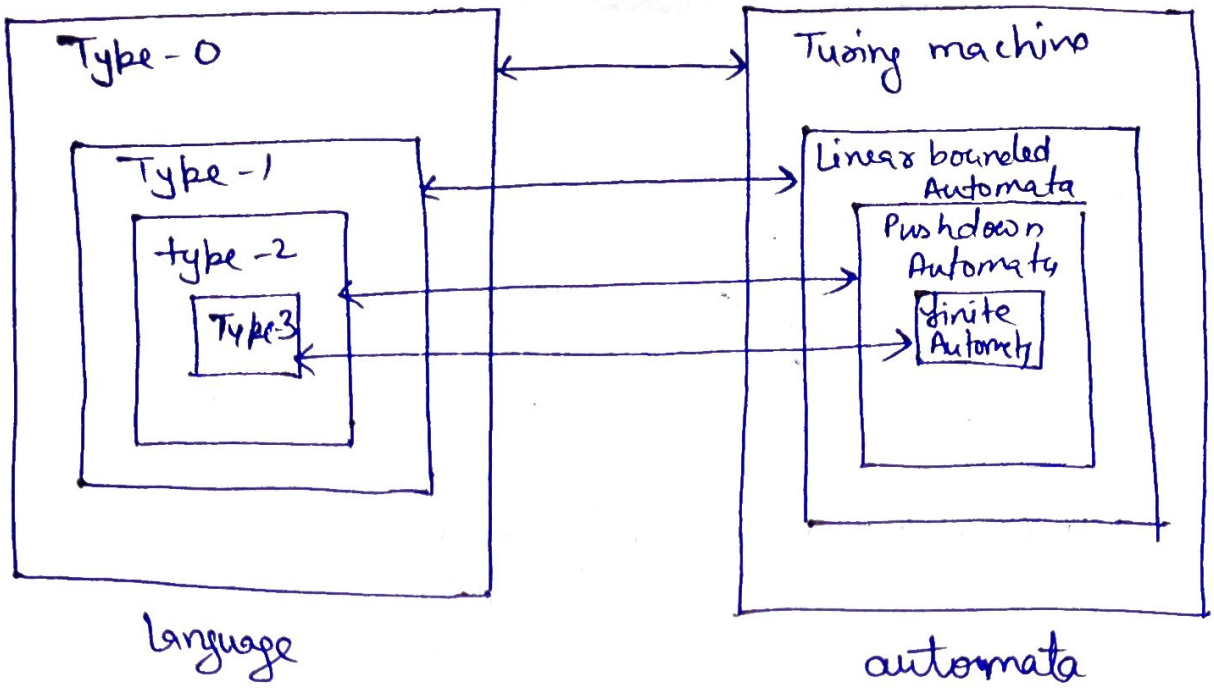
rather than giving a formal description of a program, the goal is to show the equivalence b/w a program and another notation - (predicate logic)

CHOMSKY HIERARCHY -

Chomsky describe a model for grammars -



Pragya Gaur



relationship b/w language & automata

Type-3 :- (regular grammar)

Production-

$A \to aB \mid a$

where

$a \in \Sigma$ (Terminals)

$B \in V_N$ (Non terminals)

$A \to B$ X

$A \to aB \mid a$ ✓

finite state automata and regular grammar provide model for building lexical analyzer in a translator for a programming language.

properties of regular grammars:

1) most properties of such grammar are decidable.
2) regular grammar ~~can~~ can generate string of the form α^n for any finite sequence α and any integer n.

Type-2 :- Production - (Context free grammar)

$A \to \alpha$

where $A \in V_N$ (Non terminals)

$\alpha \in (V_N \cup \Sigma)^*$

$A \to a$ ✓
$A \to AB$ ✓
$A \to aA$ ✓

Properties :-

1) many properties of such grammar are decidable.
2) these grammar can be used to count two items and compare them.
3) CFG can be implemented via stack.

Type 1 :- (Context sensitive grammar)

Production -

$\alpha \to \beta$

where $\alpha \in V_N$ (any string of non terminals)

$\beta \in (V_N \cup \Sigma)^*$ (any string of terminals & non-terminals)

with a restriction -

no. of symbols in $\alpha \leq$ no. of symbols in β

$AB \to aB$ ✓
$AB \to Aa$ ✓
$AB \to ab$ ✓
$Aa \to aBX$

properties :

1) some properties of context sensitive grammar are unknown
2) too-complex to be useful for programming language application.
3) it generate a string that need a fixed amount of memory

type 0 - (unrestricted grammar)

production -

$$\alpha \rightarrow \beta$$

where

$\alpha \in V_N$ (any string of non terminal)

$\beta \in (V_N \cup \varepsilon)^*$ (any string of terminals & non-terminals)

any phrase structure grammar without any restriction.

properties :-

1) most properties are undecidable.
2) they can be used to recognize any computable function.