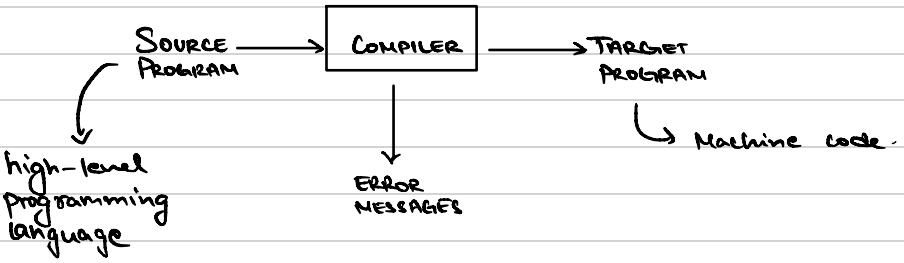
Compiler Design - I

COMPILER

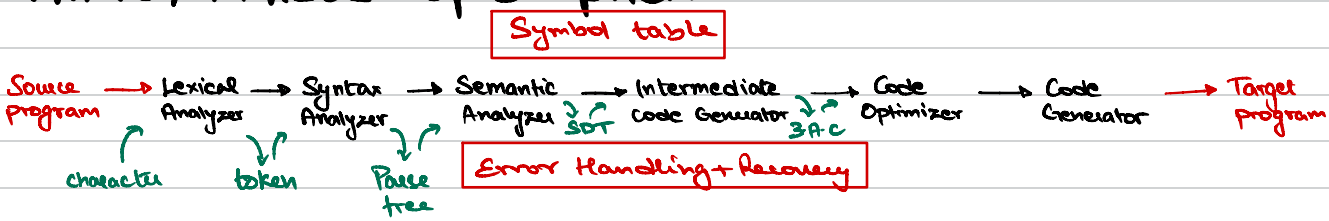
* It is a program that takes a program written in Source Language (like C, C++) & converts it into equivalent program in target language.

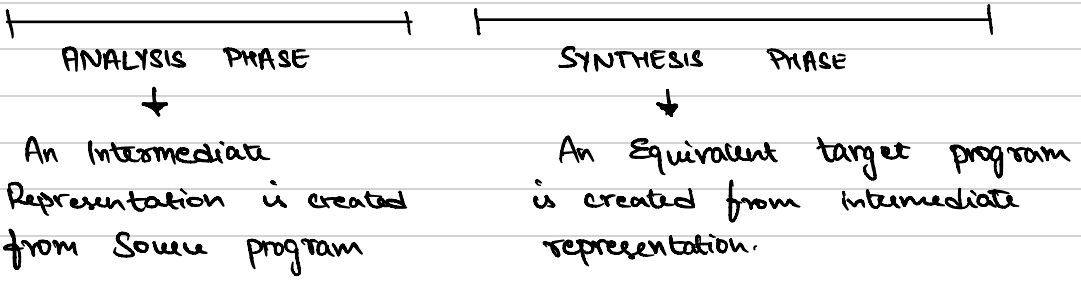


INTERPRETER:

* Instead of producing target program as a translation, Interpreter executes operations specified with inputs to produce necessary output - line-by-line.

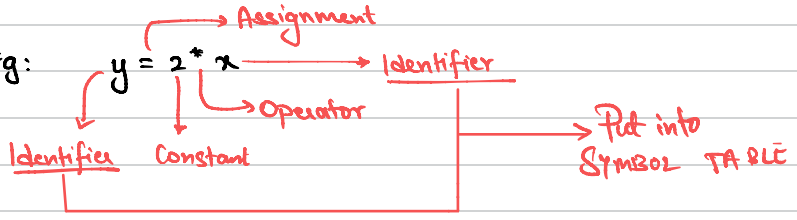
PARTS / PHASES of Compiler:





LEXICAL ANALYSIS:

* Input: Stream of characters from source program
  (gets converted into)
  Output: Stream of TOKENS / LEXEMS → Raw form of tokens
  → Pattern of Characters.
* Also called the Scanning phase
* Identifiers are put into Symbol table.

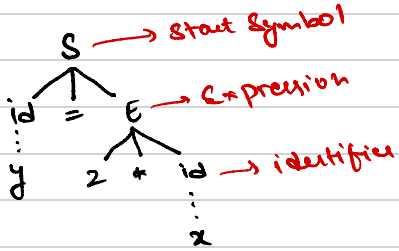


SYNTAX ANALYSIS:

↳ Parse tree

* Creates a Syntactic structure of given program
* Also called Parser.
* In a Parse tree:
  - All terminals are leaf
  - All inner-nodes are non-terminal in CFG.
* Basically, Syntax Analyzer checks if the expressions made by tokens are syntactically correct. (whether the grammar is ambiguous or not)

* Eg: $y = 2 * x$



SEMANTIC ANALYSIS:

* Checks whether the constructed Parse tree follows RULES OF LANGUAGE.
* Checks for semantic Consistency.
* Also does TYPE-CHECKING
* Basically, checks if there are any logical Error.

* Eg: $y = 2 * x$
  float ← int, int (gets converted to float)

INTERMEDIATE CODE GENERATION:

* Generation of an intermediate code in such a way that, it makes it easier to be translated into target machine code.
* Intermediate Codes are Architecture Independent.
* 3AC – generated

* Eg:
$$y = 2 * x$$
$$t_1 = 2 * x$$
$$y = t_1$$

CODE OPTIMIZATION:

* Code Optimization of Intermediate code.
* Remove unnecessary code lines, rearrange code in order to speed up Execution without wasting any Resources (CPU, Time).

CODE GENERATION:

* Optimized Intermediate Code is translated into Target code – Registers or memory location is selected for Every Variable.
* Eg: ADDF R1, R2 } Add R1, R2, Store in R1
  ADDF: float; R1: destination

LOADER: Loads relocatable Machine Code to proper location

LINK EDITOR/LINKER: Allows to make a single program from several files of relocatable machine code.

Compiler Construction Tools (P S S C D C)

#1- Parser Generator: Allows to automatically produce syntax analyzer from a grammatical programming language.

#2- Scanner Generator: Produces lexical analyzer from Regular Expression.

#3- Syntax Directed Translation Engine: Generate Intermediate Code.

#4- Code Generator: Intermediate language to Machine language

#5- Data flow Analysis engine: shows how values are transmitted from one part to another.

#6- Compiler Construction toolkit: Integrate set of Routine in different phases.

Input Buffering:

* During Lexical Analysis, to identify correct tokens, the compiler needs to look-ahead to confirm.

C-Language
look ahead after
-, = (or) < />

Fortran
Do 5 I = 1.25

Two-Buffer Scheme:

↳ Introduced to handle large look-aheads safely & efficiently.

STRUCTURE:

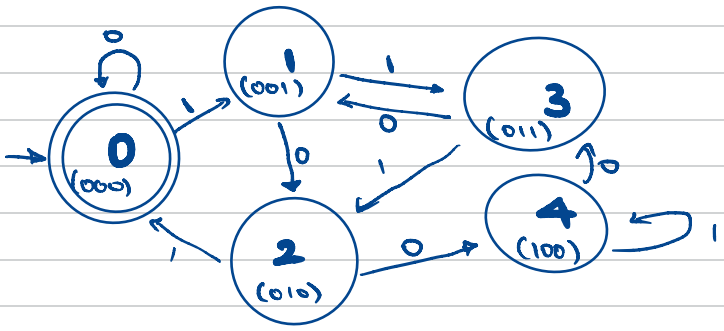
(1) Buffer1, Buffer2 (bp, fp): 2 Buffers hold the Input Stream
(2) Sentinels: Special EOF characters — Marks the end of buffers.

Eg: x == y

DFA / NFA / REGULAR EXPRESSIONS

#1- DFA that accepts numbers divisible by 5

1→ Identify Binary numbers divisible by 5 (Remainder=0)
2→ Remainders can be 0,1,2,3,4 → 5 state DFA



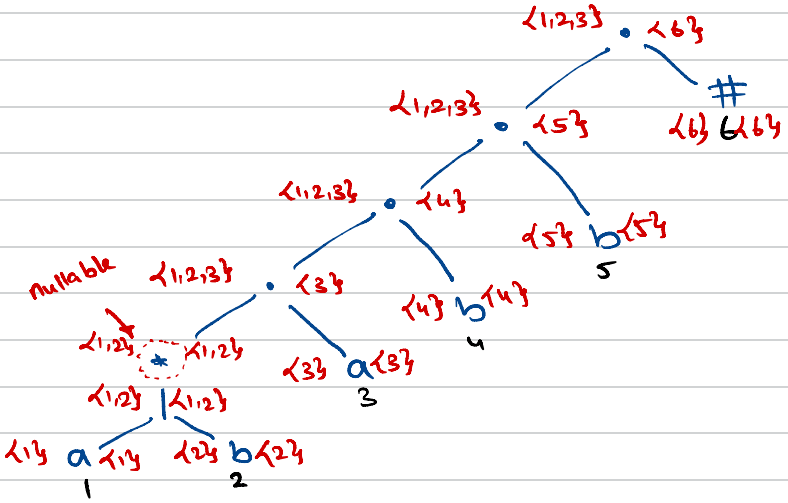
#2 Convert Regular Expression $(a|b)^*abb$ into DFA.

Node n	Nullable(n)	Firstpos(n)	Lastpos(n)
	*nullable(c1) or nullable(c2)*	*firstpos(c1) ∪ firstpos(c2)*	*lastpos(c1) ∪ lastpos(c2)*
	*nullable(c1) and nullable(c2)*	*If (nullable(c1) then firstpos(c1) ∪ firstpos(c2) else firstpos(c1)*	*If (nullable(c2) then lastpos(c1) ∪ lastpos(c2) else lastpos(c2)*
	*true*	*firstpos(c1)*	*lastpos(c1)*

1- Augmented Regex:

$(a|b)^*.a.b.b.\#$

2- first n last ↓



3- follow:

$(a|b).a.b.b.\#$
1 2 3 4 5 6

Node name	node no	follow
a	1	{1,2,3}
b	2	{1,2,3}
a	3	{4}
b	4	{5}
b	5	{6}
#	6	-

DFA: