CC Notes

Que ①: what is a compiler?

↳ It is a computer program which helps you transform source code written in a high level language into low level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code.

- It also makes the end code efficient (optimized for execution of time & memory space)

Que ②: phases of compiler construction?

↳ The process of converting the source code into machine code involves several phases or stages, which are collectively known as the phases of a compiler.

① Lexical Analysis
- 1st phase
- Reads the source code & broke into tokens, which are the basic units of programming language.
- tokens are then passed on to the next phase for further processing.

② Syntax Analysis
- 2nd phase also known as parsing.
- Takes the stream of tokens genrated by the lexical analysis phase & checks whether they conform to the grammar of the programming language.
- Output of this phase is usually Abstract Syntax tree (AST)

③ Semantic Analysis
- 3rd phase
- check whether the code is ~~syntax~~ semantically correct
- the compiler checks the meaning of the source code to ensure that it makes sense.
- compiler also checks for other semantic errors, such as undeclared variables & incorrect function calls.

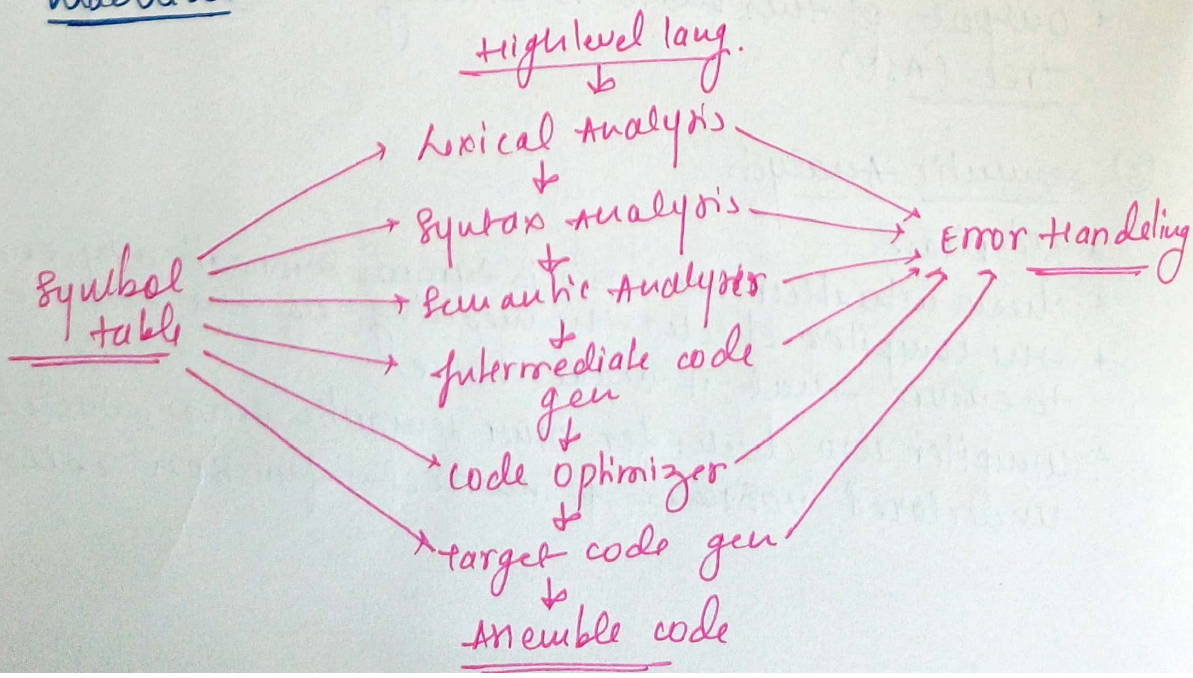
(4) Intermediate code generation
* generates an intermediate representation of the source code that can be easily translated into machine code.

(5) Optimization
* Applies various optimization techniques to the intermediate code to improve the performance of the generated machine codes.

(6) Code generation
* Takes optimized intermediate code → generates the actual machine code that can be executed by the target Hardware.

(7) Symbol Table
* Data structure being used and maintained by the compiler, consisting of all the identifier's names along with their types.
* It helps the compiler to function smoothly by finding the identifiers quickly.

(8) Target code generator
* To write the code that machine can understand and also register Allocation, instruction selection
* the output is dependent on the type of assembler.
* This is the final stage of compiler
* Optimized code then converted into → Relocatable machine

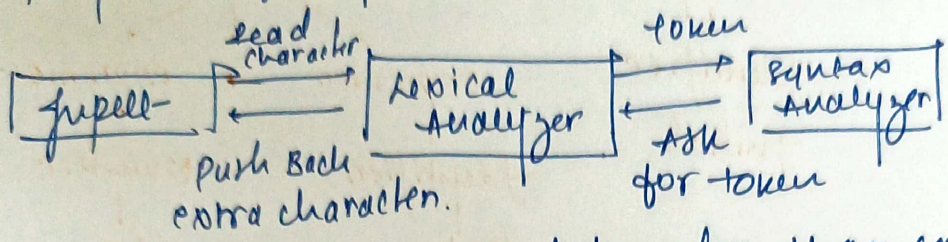


Q(3): Explain Lexical Analyzer? 2

↳ Lexical Analyzer is the first phase of the compiler also known as a scanner. It converts High level input program into a sequence of tokens.

① Can be implemented with the Deterministic finite Automata

② The output is the sequence of token that is sent to the parser for syntax Analysis.



Token → It is a sequence of character that can be treated as a unit in the grammar of the programming language.

* type token (id, number, Real...)
* Punctuation token (IF, void, Return...)
* Alphabetic tokens (keywords)

Lexeme → The sequence of character matched by a pattern to form the corresponding token or a sequence of input character that comprises a single token

Eg. "float", "abs_zero_kelvin", "=", "_"

Q(4): Explain Analysis of the source program?

↳ In compiling, analysis consist of three phases:

① Lexical Analysis
② Syntax Analysis
③ Semantic Analysis.

① Lexical Analysis

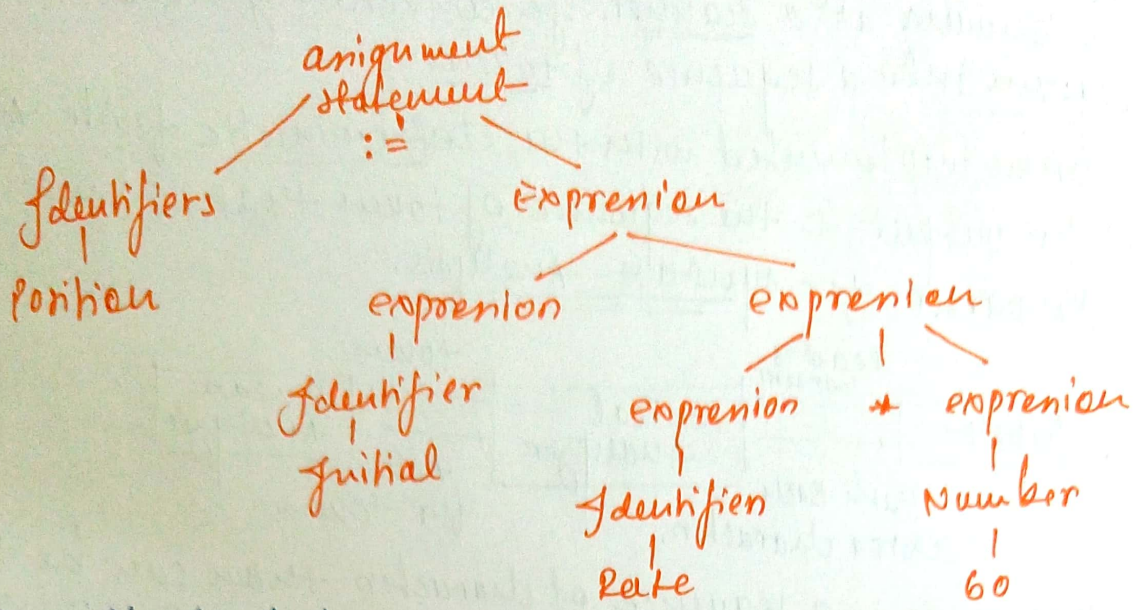
Example - position = initial + rate * 60

Identifiers - position, initial, rate

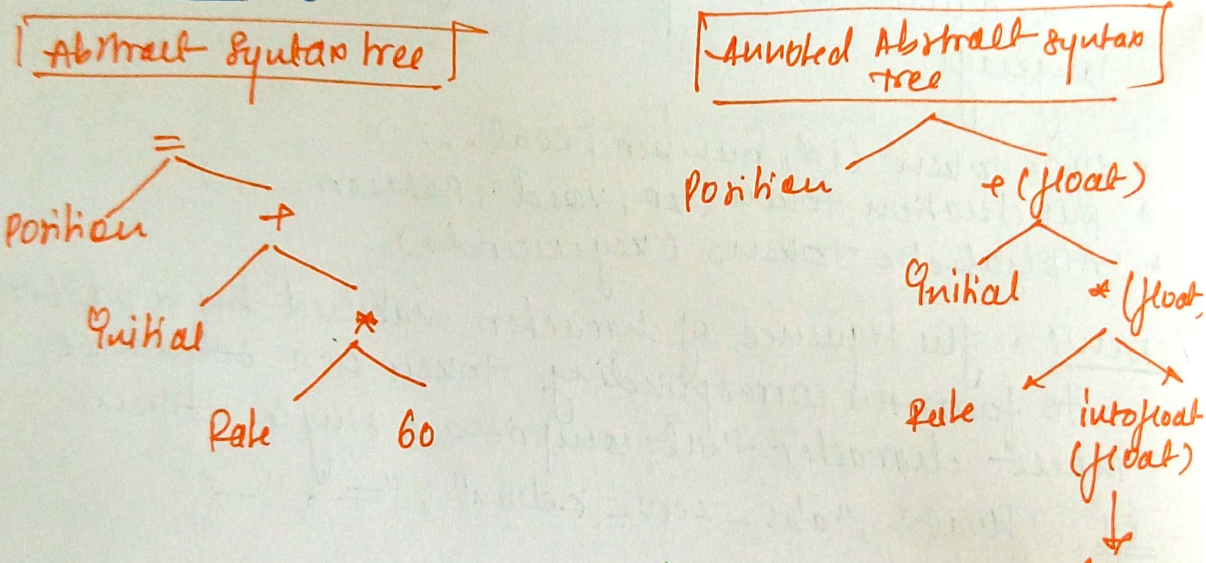
Operator - +, *

Assignment Symbol - =

Number - 60
Blanks - eliminated



② Semantic Analysis

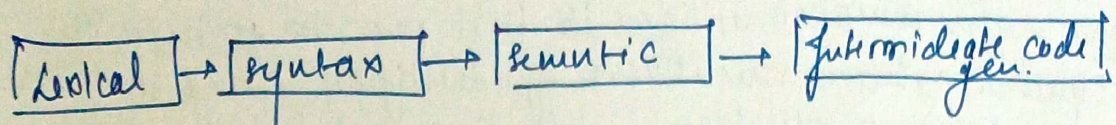


Ques ⑤: Grouping of phases in Compiler ?

↳ Generally phases are divided into 2 parts:

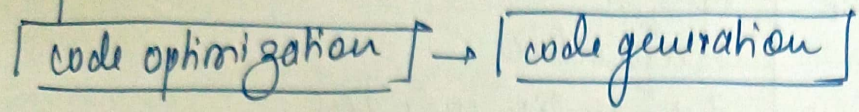
① Front END phases - The front end consist of those phases or parts of phases that are source language - dependent and target machine, independents.

- This are generally consist of lexical analysis, semantic analysis, syntactic analysis, symbol table creation & intermediate code generation.
- A little part of code optimization can also be included in the front-end part.
- It also includes the error handeling that goes along with each of the phases.



② Back END Phase – The portions of compiler that depend on the target machine & do not depend on the source language are included in the back end.

- In code generation, and necessary features of code generation optimization phases, along with error handling & symbol table operation are also included.



✱ Panes in compiler

A pan is a component where parts of one or more phases of the compiler are combined when a compiler is implemented.

2 panes – ① one pan ② two pan

① one pan - All phases are grouped into one phase. The six phases are included here in one pan.

② two pan - In two pan the phases are divided into 2 parts.

Analysis — Front part

③ one pan – It generates a structure of machine instruction and then sum up with their machine address for these guidlines to a rundown of directions to be backpatched once machine address is generated.

- It used to pan program for one time.
- Whenever the line source is handeled, it is checked & the token is removed.

Lexical
↓
syntax
↓
Semantic
↓
intermediate codegen
↓
code optimization
↓
code generation.

④ **two pass** - the compiler utilizes its first pass to go into its symbol table a number of identifier along with the memory areas to which these identifiers relate.

In second pass the compiler can read the result document delivered by the first pass, assemble the syntatic tree and deliver the syntactical examination.

Front end phase [Lexical + Syntax + Semantic + Intermediate code gen

Backend phase [code optimization + code generation

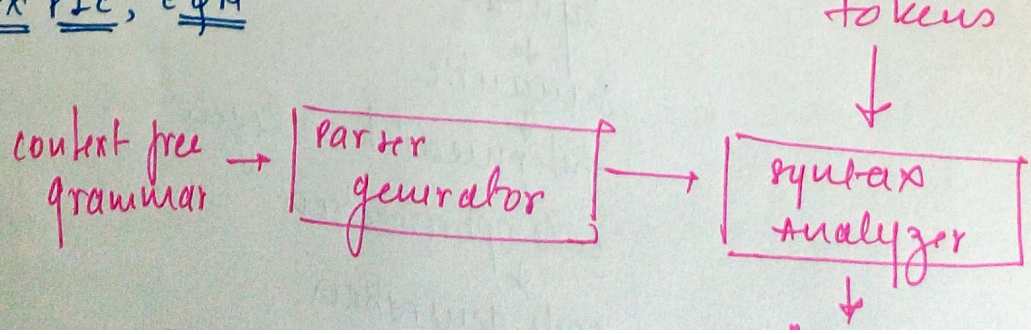
- grouping into a single phase make the arrangement quicker as the compiler isn't expected to move ~~out~~ to various modules or phases.
- Reducing no. of passes is inversely proportional to the ↑ time efficiency for reading from & writing to intermediate files can be diminished.

Ques ⑤: Explain compiler construction tools.

① **Parser Generator:** It produces syntax analyzer (parser) from the input that is based on a grammatical description of programming languages or on context-free grammars.

It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.

Ex PIC, EQN



② Scanner Gen...

descript...
It generates
Expression.
Ex Lex

Specificati...
Regular Exp...

③ Syntax di...
↳ It gen...
at the ...
This engi...
the prod...
Each code
transla...

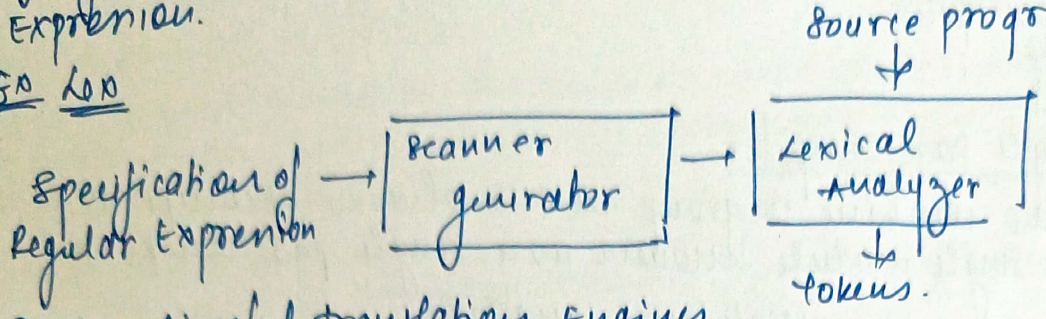
④ Autom...
lang...
Each
using
input

⑤ Data-...
Dat...
that
are

⑥ Comp...

② Scanner Generator - It generates lexical Analyzer from the input that consist of regular expression description based on tokens of a language.
It generates finite automation to recognize the regular expression.

Ex: Lex



③ Syntax directed translation Engines
↳ It generates intermediate codes with three address format at the input that consist of a parse tree.
These engines have routines to traverse the parsetree & then produce the intermediate code.
Each code of parse tree is anociated with one or more translation.

④ Automatic code generation - It generates the machine language for a target machine.
Each operation of a intermediate language is translated using a collection of rules & then is taken as an input by the code generator.

⑤ Data-flow Analysis Engines - It is used in code optimization
Data flow analysis is a key part of the code optimization that gathers the info that is the values that flow from one part to another.

⑥ Compiler construction toolkit - It provides an integrated set of routines that aids in building compile components or in the constructions of various phases of compiler.

ques ⑥: One pass compiler?

↳ A compiler pass refer to the traversal of a compiler through the entire program. compiler passes are of two types Single pass compiler, and two pass compiler or Multi-pass compiler.

Types

① Single pass compiler

If we combine or group all the phases of compiler design in a single module known as a single pass compiler.

High level language
↓
lexical Analyzer
↓
Syntax Analyzer
↓
Semantic Analyzer
↓
Intermediate code gen
↓
code optimization
↓
code generation
↓
Low level language.

In above diagram, there are all 6 phases are grouped in a single module, some points of the single pass compiler are as:

- A one pass / single pass compiler is a type of compiler that passes through the part of each compilation unit exactly once.
- It is faster and smaller than Multi pass compiler
- A ~~it~~ It is one that process the input exactly once, so going directly from lexical Analysis to code generator, and then going back for the next read.

Problem

- cannot optimize very well due to the context of expression are limited.

- We can't back
be limited to
- command int
as single pass

ques 7: A Role of

- It is the first
operate as a
phases of a
- Reads input
into lexeme
lexeme.
- token sent
- If it is loc
an inter mec
parser wou
- Eliminate
- Interacts
to the par
- It seperat
into group
- Also hand
and whi

ques 8: Sup

↳ It is a
to the
the sou

In many
a time

It is a
input
& reduc

① The b
of in
the p
bloc

- We can't back up and procen it again so grammar should be limited & simplified.
- command interpreter such as bash / sh can be considered as single pan compiler.

ans 7: A Role of the lexical Analyzer?

- It is the first phase of the compiler where a lexical analyzer operate as an interface betn the source code & rest of the phases of a compiler.
- Reads input character of the source program, group them into lexemes & produce sequence of token for each lexeme.
- token sent to parser for syntax Analysis.
- If it is located in the seperate pan of compiler it can need an intermediate file to locate its output from which the parser would takes its input.
- Eliminate the need for the intermediate file.
- Interacts with the symbols table while paning token to the parser.
- It seperates the characters of the source language into groups that logically belong together, called tokens.
- Also handles issues including stripping out the comments and whitespaces.

ans 8: Input Buffering in cc?

↳ It is an imp concept in compiler design that refer to the way in which the compiler reads input from the source code.

In many cases compiler reads input one character at a time, which can be slowed & inefficient procen.

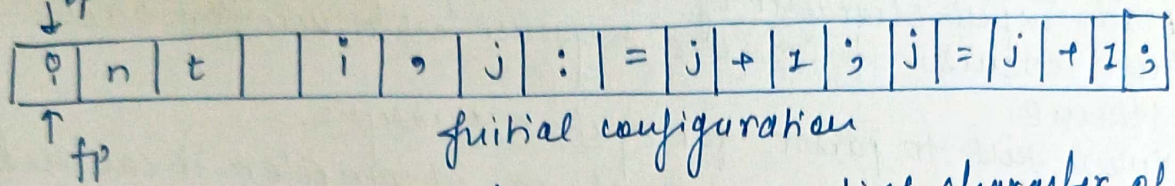
It is an technique that allows the compiler to read input in larger chunks which can improve performance & reduced overhead.

① The basic idea behind input buffering is to read block of input from the source code into a buffer., and the procen that buffer before reading the next block.

the size of buffer can vary depending on the specific need of compiler.

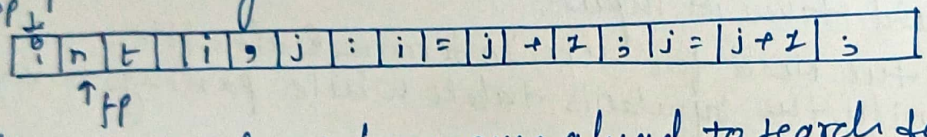
Ex A compiler for a High level programming language may use a large buffer than a compiler for a low level language.

(2) one of the main advantage of Input buffering is that it can reduce the number of system calls required to read Input from the source code.



Initial configuration

Initially both the pointer point to the first character of the input string.



- The forward pointer moves ahead to search for each of lexeme.
- As soon as blank space is encountered, it indicates end of lexemes.
- In above eg as soon as ptr (fp) encounters a blank space the lexeme "int" is identified.
- The fp will move ahead at a white ~~pointer~~ space, when fp encounters white space it ignores & moved ahead. then both the begin ptr (bp) and forward ptr (fp) are set at next token.
- The input character thus read from secondary storage but is stored.
- There are 2 method — (1) one Buffer scheme
  (2) two Buffer scheme

(1) one Buffer - In this scheme, only one Buffer is used to store the input string but the problem with this scheme is that if lexeme is very long then it crosses the buffer boundary.
to scan rest of the lexeme the buffer has to be refilled, that makes overwriting the first of lexeme.

(2) Two Buffer sc
in
the input
alternately

Adv - (1) can
reac
(2) can

Disad - (1) si
(2) ff

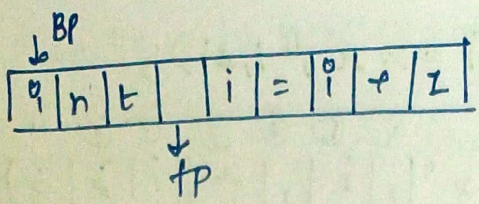
Ques: Speci

Tokens - A
prog
be fur

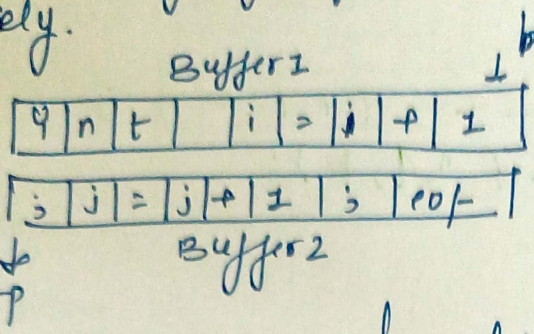
Types

keywords
- if, while
else, for

Exampl
Int
if
els
}



② Two Buffer Scheme – to overcome the problem of one Buffer in this method two Buffers are used to store the input string. The first & second Buffer are scanned alternately.



two Buffering scheme storing input string.

Adv – ① can Reduce the number of system calls required to read input from the source code. → improve performance.

② can simplify design of compiler

Disad – ① size of Buffer too large → may consume too much memory.

② If Buffer is not managed properly → can leads to error

Ques 9: Specification of token in CC?

Tokens – A token is the smallest individuals element of a program that is meaningful to the compiler. It cannot be further broken down.

Types

keywords	identifiers	operators	constant	Seperator	Special symbol
- if, while, else, for	- a, b, x	- <, >, =, ==	20, 30, 10	<, >, ;, (,) ...	@, $, #

Example

```
int main (int a, int b)
{
  if (a<b)
    return (a);
  else
    return b;
}
```

if, int – keyword
main – identifier
(→ seperator
a, b → identifier.

Example - printf("i=%d, &i=%x", i, &i);

① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ 10

| printf | (| "i=%d, &i=%x" | , | i | , | & | i |) | ; |

total tokens - 10.

Example

```
main()
{
a=b+++---+++==;
printf("%d%d ",a,b);
}
```

| main | (|) |

| { |

| a | = | b | ++ | + | - | -- | - | ++ | + | = | = | ; |

| printf | (| " | %d | %d | " | , | a | , | b |) | ; |

| } |

total token - 25

A Language specifying Lexical Analyzer.

↳ There is a wide range of tools for construction Lexical Analyzer.

① Lex

② YACC

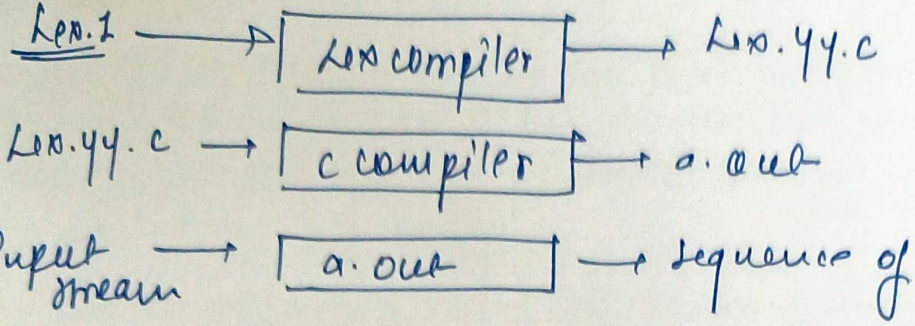
Lex is a computer program that generates lexical Analyzer.

It is commonly used in the YACC parser generator.

Creating lexical Analyzer

- First, a specification of a lexical Analyzer is prepared by creating a program Lex.l in the lex language. Then Lex.l is run through the lex compiler to produce a C program Lex.yy.c.

- finally, Lex.yy.c is run through the C compiler to produce an object program a.out which is the Lexical Analyzer that transforms an input stream into a sequence of tokens.



(*) Lex specification

It consist of 3 parts. { definations }
{ Rules }
{ user subroutines }

Definition includes declaration of variables, constants, & Regular definitions.

Rules are statement of the form

p_1 {action1}
p_2 {action2}
p_3 {action3}
:
p_n {actionn}

where p_i is regular expression and actioni describes what action the lexical Analyzer should take when pattern p_i matches a lexeme.

user subroutines - are auxiliary procedures needed by the actions. These can be compiled seperately & loaded with the lexical Analyzer.

(*) YACC - yet another compiler compiler

↳ It provides a general tools for describing the input to computer program. The YACC user specifies the structure of his input.

YACC turns such a specification into a subroutines that handles the input process

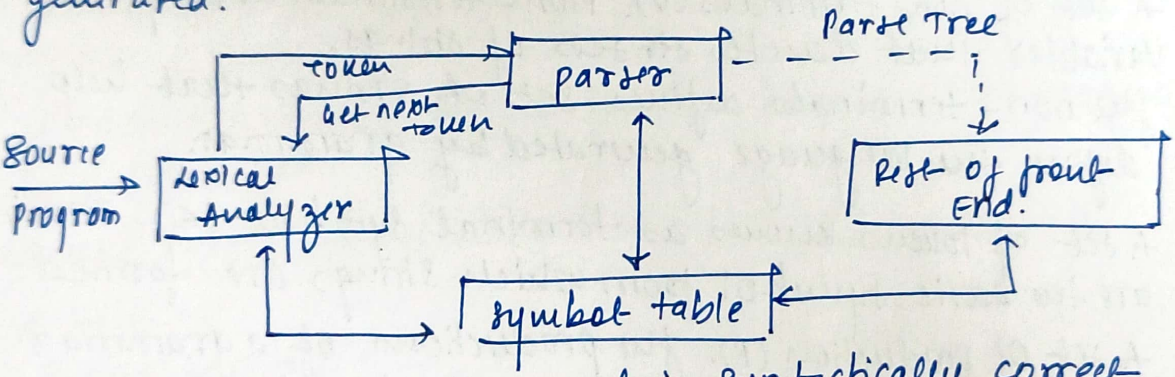
Finite Automata — (1) Non-deterministic Finite Automata (NFA)
(2) Deterministic Automata. (DFA)

UNIT-2

ques 1 Role of parser?

↳ parsing is performed at the syntax analysis phase where a stream of token is taken as input from the lexical analyzer and the parser produces the parse tree for the tokens

- In syntax analysis phase, a compiler verifies whether or not the token generated by the lexical analyzer are grouped according to the syntatic rules of the language.
  ↳ this is done by parser
- The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be grammar for the source language.
- It detects & reports any syntax errors and produces a parse tree from which intermediate code can be generated.



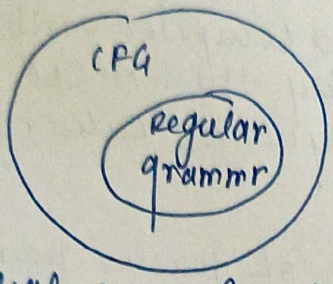
- to ensure that the source code is syntactically correct
- performs error Recovery.

ques 2 what is grammar?

↳ A set of instructions about how to write statements that are valid for that programming languages.

- We have seen that a lexical analyzer can identify tokens with the help of regular exprenion & pattern rules.
- But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular exprenions.

- But a Regular expression cannot check balancing tokens, such as parenthesis. Therefore this phase uses context-free grammar (CFG) which is recognized by push-down automata.
- CFG on the other hand, is superset of Regular grammar



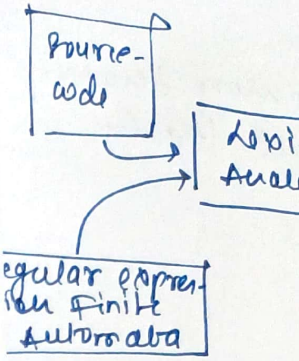
- It implies that every Regular grammar is also context free, but there exists some problems, which are beyond the scope of Regular grammar.

#CFG

- A set of non-terminals (V). Non-terminals are syntatic variables that denotes sets of strings. The non-terminals defines sets of strings that help define the language generated by grammar.
- A set of tokens known as terminal symbol (Σ). Terminals are the basic symbol from which strings are formed.
- A set of production (P). The productions of a grammar specify the manner in which the terminals & non-terminals can be combined to form strings.
- One of the non-terminals is designed as the start symbol (S).

Syntax Analyzer

- A syntax Analyzer or parser takes the input from a lexical analyzer in the form of tokens streams.
- The parser analyzes the source code against the production rules to detect any error in the code.
- The output of this phase is a parse Tree.



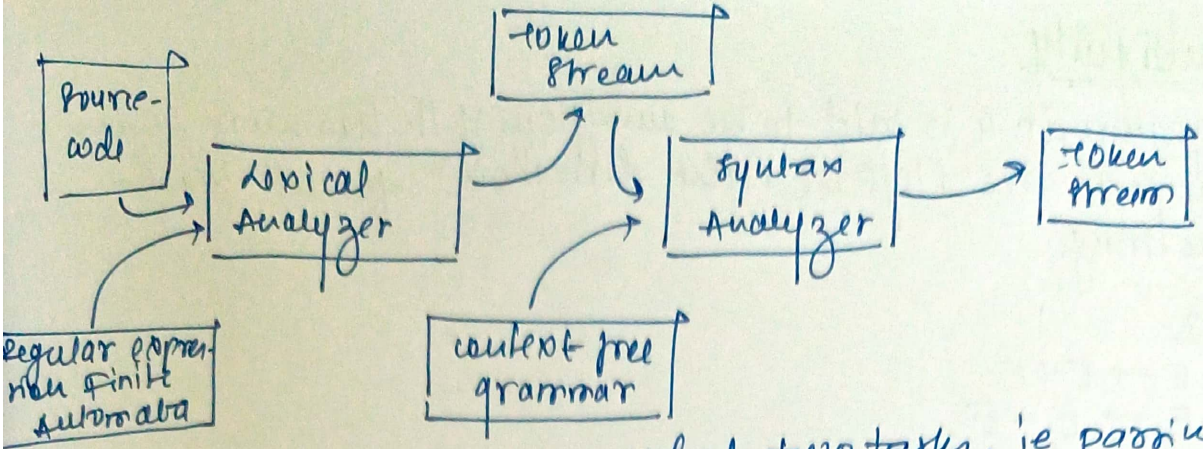
- This way, the code, as the Output
- Parser are errors exist

#Parse tree

A parse tree convenient Symbol. The start the parse We take the

E → E*E
E → E
E → id
E → id
E → id

- All leaf terminals
- All inter non-t
- In order gives or



- This way, the parser accomplished two tasks, ie parsing the code, looking for errors and generating a parse tree as the output of the ~~parser~~ phase.
- parser are expected to parse the whole code even if some errors exist in the program.

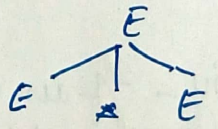
Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol.

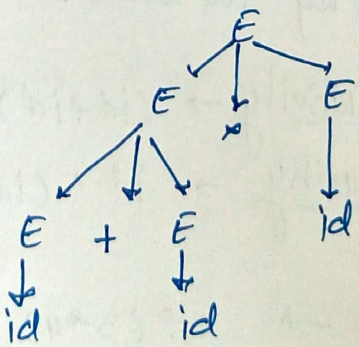
The start symbol of the derivation becomes the root of the parse tree.

We take the left-most derivation of a+b*c

$E \to E*E$
$E \to E+E*E$
$E \to id+E*E$
$E \to id+id*E$
$E \to id+id*id$



$E \to E*E$



- All leaf nodes are terminals
- All interior nodes are non-terminals
- In order traversal gives original input string.

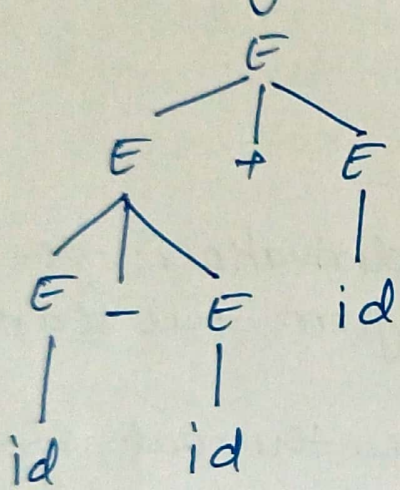
Ambiguity

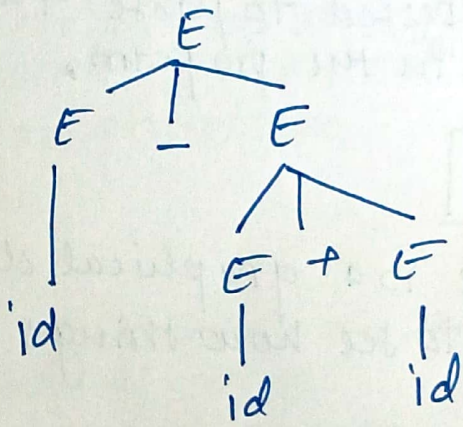
A grammar G is said to be ambigious if it has more than one parse tree (left or right derivation) for at least one string

Ex

$E \rightarrow E+E$
$E \rightarrow E-E$
$E \rightarrow id$

for grammar id+id-id the above grammar generates two parse tree.





The Ambiguity can be removed by 2 methods

Associativity Precedence.

① Associativity - If an operand has operator on both sides, the sides on which the operator takes this operand is decided by the associativity of those operator.

Right Associativity → (id+id)+id

Left Associativity → id^(id^id)

② Precedence → 2+(3*4)

First & Follow

question ①

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow id \mid (E)$

First -

$E \rightarrow \{+\} \rightarrow \{id, \epsilon\}$
$E' \rightarrow \{+, \epsilon\}$
$T \rightarrow \{F\}$
$T' \rightarrow \{*, \epsilon\}$
$F \rightarrow \{id, \epsilon\}$

Follow -

$E \rightarrow \{\$,)\}$
$E' \rightarrow$ Follow(E) $\rightarrow \{\$,)\}$
$T \rightarrow$ first(E') $\rightarrow \{+, \$,)\}$
$T' \rightarrow$ follow(T) $\rightarrow \{+, \$,)\}$
$F \rightarrow$ First(T') $\rightarrow \{*\} \cup$ Follow(T) $\cup$ follow(T') $\rightarrow \{*, +, \$,)\}$

ques **First & Follow**

	first()	Follow()
$S \rightarrow ABCDE$	$\{a, b, c\}$	$\{\$\}$
$A \rightarrow a \mid \epsilon$	$\{a, \epsilon\}$	$\{b, c\}$
$B \rightarrow b \mid \epsilon$	$\{b, \epsilon\}$	$\{c\}$
$C \rightarrow c$	$\{c\}$	$\{d, e, \$\}$
$D \rightarrow d \mid \epsilon$	$\{d, \epsilon\}$	$\{e, \$\}$
$E \rightarrow e \mid \epsilon$	$\{e, \epsilon\}$	$\{\$\}$

Example

	First	Follow
$S \rightarrow Bb \mid Cd$	~~$\{B, C\}$~~ $\Rightarrow \{a, b, c, d\}$	$\{\$\}$
$B \rightarrow aB \mid \epsilon$	$\{a, \epsilon\}$	$\{b\}$
$C \rightarrow cC \mid \epsilon$	$\{c, \epsilon\}$	$\{d\}$

Context free grammar

↳ Formal grammar
– Has 4 tuples (V, T, P, S)

V – collection of variables or non-terminal symbols
T – Set of terminals
P – production Rules
S – starting symbol.

A grammar said to be context free grammar if every production is in the form of:

$G \rightarrow (VUT)^*$, where $G \in V$

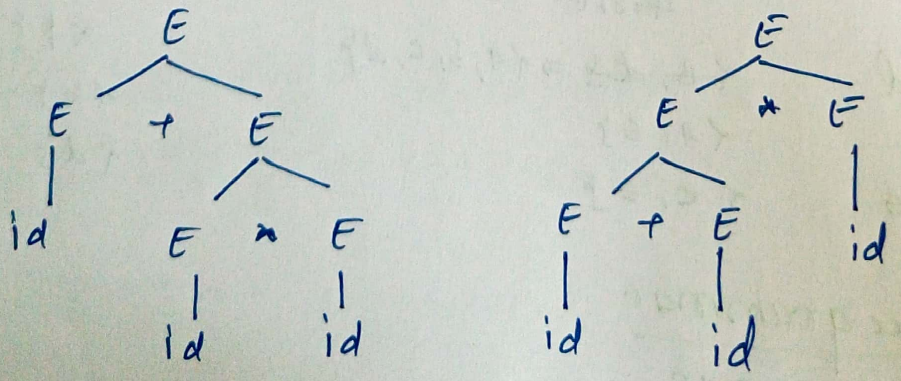
- And left side of the G, here in the example, can only be a variable, it cannot be a terminal.
- But on the right-hand side here it can be a variables or terminals or both combination of variable & terminals.

Types of grammar

- On Basis of Production Rule
  - Type 0
  - Type 1
  - Type 2
  - Type 3
- on basis of derivation tree
  - Ambiguous
  - unambiguous
- on basis of string.
  - Recursive
  - Non-Recursive.

check whether grammar is ambigious or not

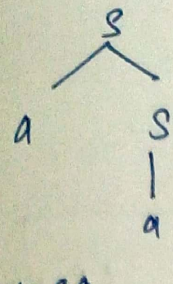
① $E \rightarrow E+E \mid E*E \mid id$



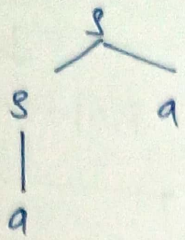
– Ambigious (As 2 parse tree are created).

① $S \to as \mid Sa \mid a$ $\quad w = aa$

String to be obtain



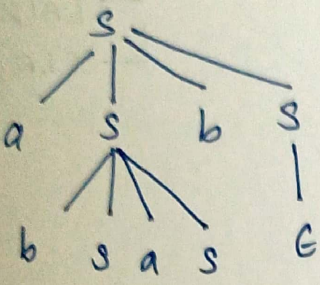
$w = aa$



$w = aa$

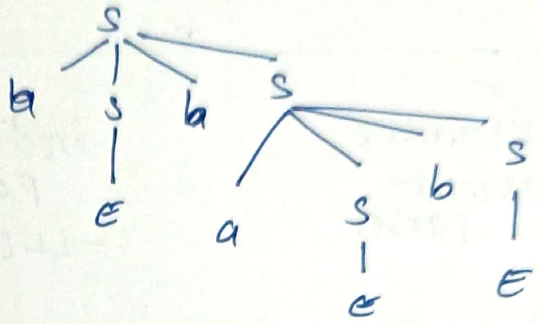
— Ambigious grammar.

② $S \to asbs \mid bsas \mid \epsilon$ $\quad w = abab$



$w = abab$

— Ambigious



$w = abab$

first & follow Example

① $S \to ABC \mid ghi \mid jkl$
$A \to a \mid b \mid c$
$B \to b$
$D \to d$

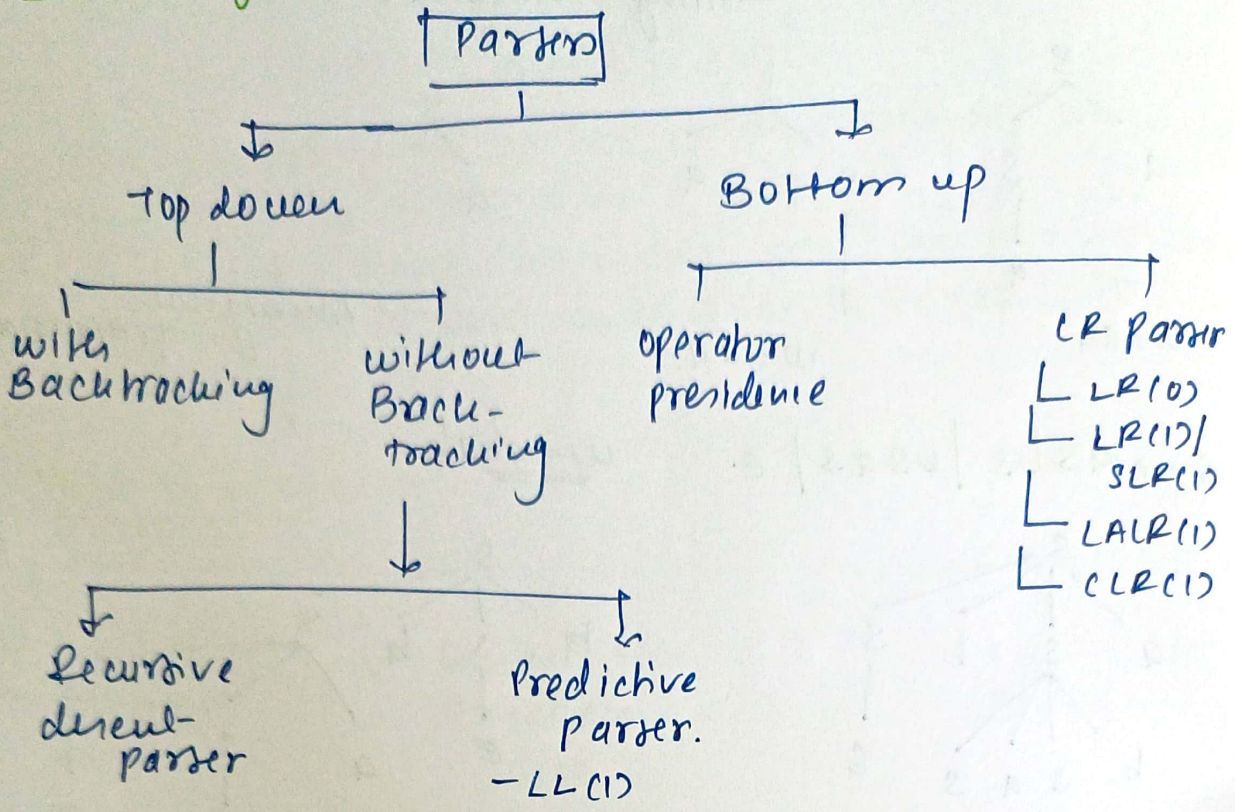
$\text{first}(D) = \{d\}$
$\text{first}(B) = \{b\}$
$\text{first}(A) = \{a, b, c\}$
$\text{first}(S) = \{A, g, j\}$
$= \{a, b, c, g, j\}$

② Follow

$S \to ACD$
$C \to a \mid b$

$\text{Follow}(S) = \{\$\}$
$\text{follow}(A) = \text{first}(C)$
$= \{a, b\}$
$\text{follow}(C) = \text{first}(D)$
$= \{d\}$
$\text{follow}(D) = \text{follow}(S)$
$= \{\$\}$

Parsing



① Top down parsing

- It is based on left most Derivation.
- The proces of constructing the parse tree which start from the root & goes to the leaf is top-down parsing

1. top down parser construct from the grammar which is free from ambiguity & left Recursion
2. It use left most derivation to construct a parse tree.
3. It does not allow grammar with common prefixes.

② Recursive descent parser

- It is a type of top down parser
- The parser which is using recursive function or procedure to parse any given string is known as Recursive descent parser.
- without involvement of Backtracking.
- It uses different Recursive procedures for procesing the given string, with the help of recursive languages.

, fw ... of this parsing is relating every non terminal with their corresponding procedures.

Limitation
① cannot parse left Recursive or left factored grammar
② aren't fast as other parsing method
③ quite difficult to produce good or meaningful error menages

Code

```
E → iE'
E' → +iE' | E

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

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

```
match (char c)
{ if (l == c)
  l = getchar();
  else
  printf ("Error");
}

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

② Predictive parser:

- It is a recursive decent parser with no backtracking or backup.
- Top-down parser
- At each step, the choice of the rule to be expanded is made upon the next terminal symbol.

to check predictive parsing
① Elimination of left Recursion
② Left factoring
③ first & follow functions
④ Predictive parsing table
⑤ Parse the input string

grammar – $E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

① Eliminate left Recursion

$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid id$

② Left factoring → To check if two left symbol of two another symbol in same string is same.

③ First & Follow

First (E) = {T} = {F} = {(, id}
First (E') = {+, ε}
First (T') = {*, ε}

Follow

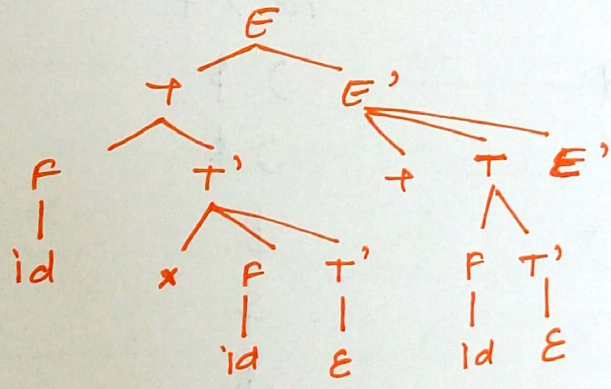
Follow (E) = {$,)}
follow (E') = {$,)}
follow (T) = {+, $,)}
follow (T') = {+, $,)}
follow (F) = {*, +, $,)}

④ Predictive parsing table.

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

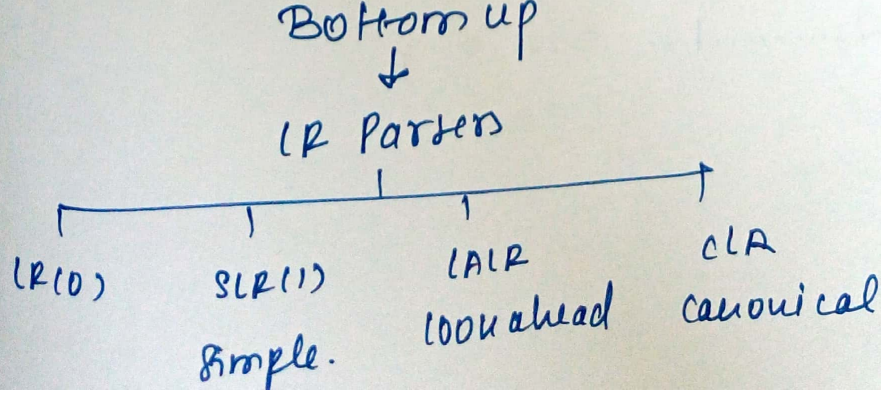
Stack	Input	Output
$E	id * id + id $	
$E'T	id * id + id $	E → TE'
$E'FT'	id * id + id $	T → FT'
$E'Tid	id * id + id $	F → id
$E'T'	* id + id $	
$E'TF*	* id + id $	T' → *FT'
$E'T'id	id + id $	F → id
$E'	+ id $	T' → ε
$E'T+	+ id $	E' → +TE'
$E'TF+	id $	T → FT'
$E'T'id	id $	F → id
$E'	$	T' → ε
$	$	

Parse tree



2) Bottom up parser

It is defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the right most derivations of w in reverse.



Que $E \to E+E$
$E \to E*E$
$E \to (E)$
$E \to id$

String: id * (id + id)

Stack	Input	Action
$	id * (id+id)$	shift id
$id	* (id+id)$	E → id
$E	* (id+id)$	shift *
$E*	(id+id)$	shift (
$E*(	id+id)$	shift id
$E*(id	+id)$	~~shift +~~
$E*(id	+id)$	~~shift id~~ E → id
$E*(E	+id)$	shift +
$E*(E+	id)$	shift id
$E*(E+id	)$	E → id
$E*(E+E	)$	shift)
$E*(E+E)	$	~~shift~~ E → E+E
$E*(E)	$	E → E*E
$E*E	$	E → (E)
$E	$	E → E*E
	($)	

In input string only $ is left then if then only string will get accepted.

③ operator precedence parsing.

- Stack — Input
  $ — w$
- terminal on top of the stack is a, symbol pointed to by input pointer is b
  (i) $a \lessdot b$ or $a \doteq b$
  shift
  (ii) if $a \gtrdot b$
  pop, until top of stack terminal is $\lessdot$ to the terminal most recently popped.

- Stack — Input
  $S — $
  ↳ completed successfully.

table — grammar — $S \to a \mid \uparrow \mid (T)$
$T \to T, S \mid S$

	a	↑	(	)	,	$
a				$\gtrdot$	$\gtrdot$	$\gtrdot$
↑				$\gtrdot$	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\lessdot$	$\doteq$	$\lessdot$	
)				$\gtrdot$	$\gtrdot$	$\gtrdot$
,	$\lessdot$	$\lessdot$	$\lessdot$	$\gtrdot$	$\gtrdot$	
$	$\lessdot$	$\lessdot$	$\lessdot$			

Stack	Input	Action
$	(a,a)$	$ $\lessdot$ (| shift
$(	a,a)$	($\lessdot$ a | shift
$(a	,a)$	a $\gtrdot$, | POP(a) reduced by $S \to a$
$(S	,a)$	Reduce $\uparrow \to S$
$(T	,a)$	($\lessdot$, shift

$(T, a) $
$ (T, a) $
$ (T, S) $

$ (T) $

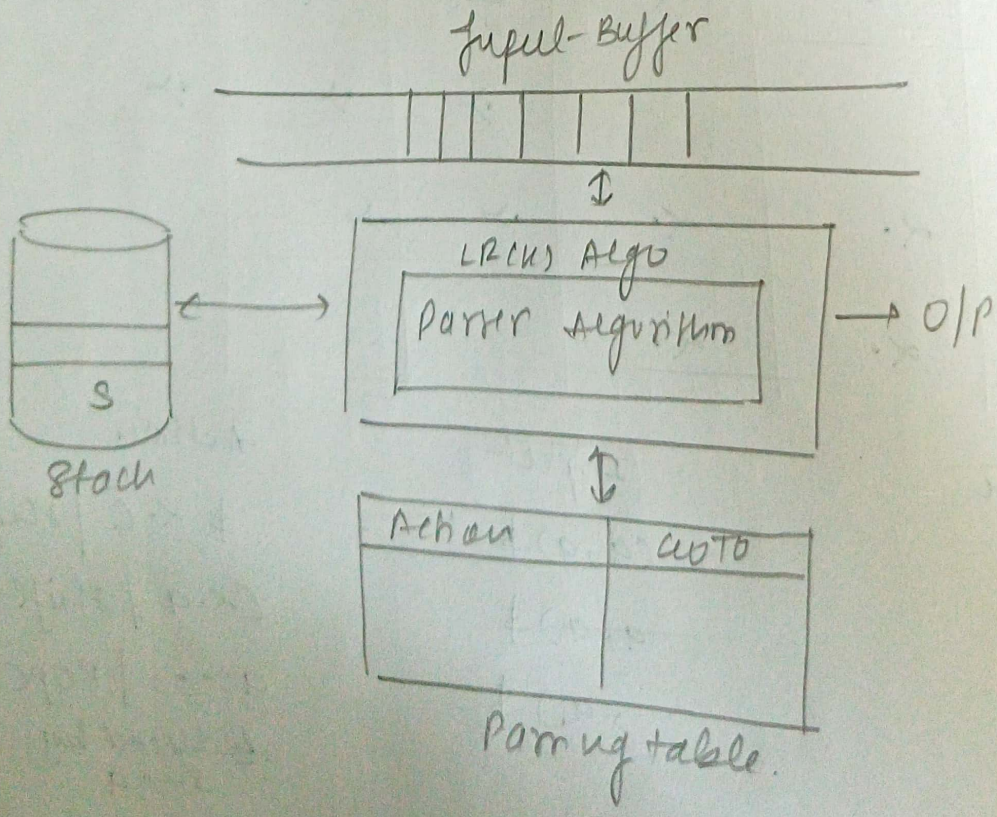
$ S $

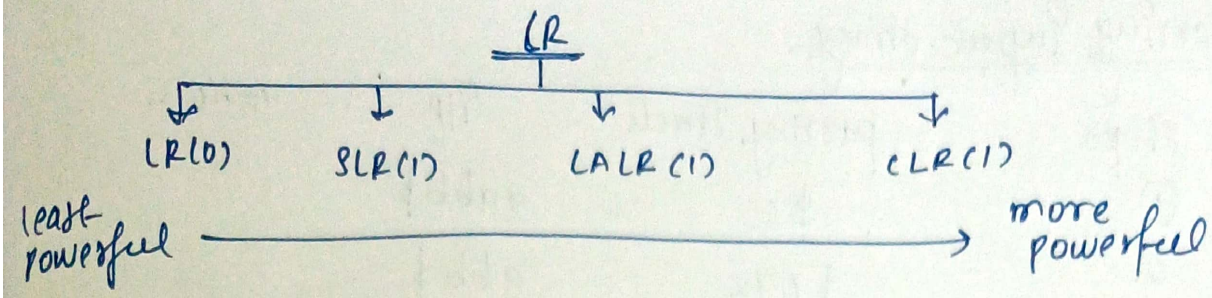
a) $: a shift
a .>), pop (a)
reduce S → a
. :>) pop T, S
reduce by
T → T, S
(=) , shift
) .> $, pop (T
reduced
S → (T)

Accept.

(4) LR Parsers

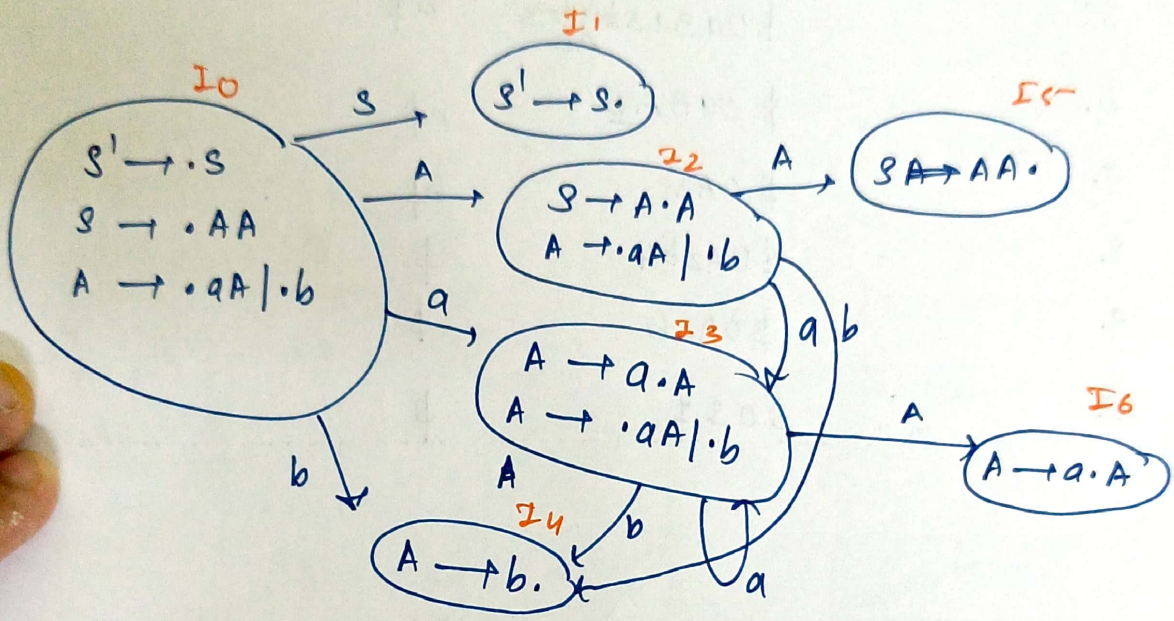
- Bottom up parser for context free grammar
- Most powerful parser
- L Refer to left to right scanning
- R Refer to Rightmost derivation.





① LR(0)

Example $S \rightarrow AA$
$A \rightarrow aA \mid b$

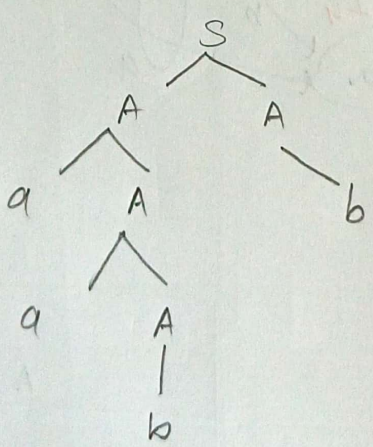


Table

States	Action			GOTO	
	a	b	$	A	S
I_0	s_3	s_4		2	1
I_1			accept		
I_2	s_3	s_4		5	
I_3	s_3	s_4		6	
I_4	r_3	r_3	r_3		
I_5	r_1	r_1	r_1		
I_6	r_2	r_2	r_2		

Parsing Input string.

Steps	parsing stack	i/p	Action.
1.	$0	aabb$	shift a_3
2.	$0a3	abb$	shift a_3
3.	$0a3a3	bb$	shift b_4
4.	$0a3a3b4	b$	reduce r_3 ($A \to b$)
5.	$0a3a3A6	b$	Reduce r_2 ($A \to aA$)
6.	$0a3A6	b$	Reduce r_2 ($A \to aA$)
7.	$0A2	b$	shift b_4
8.	$0A2b4	$	Reduce r_3 ($A \to b$)
9.	$0A2b	$	reduce r_1 ($S \to AA$)
10.	$0S1	$	Accepted



→ aabb

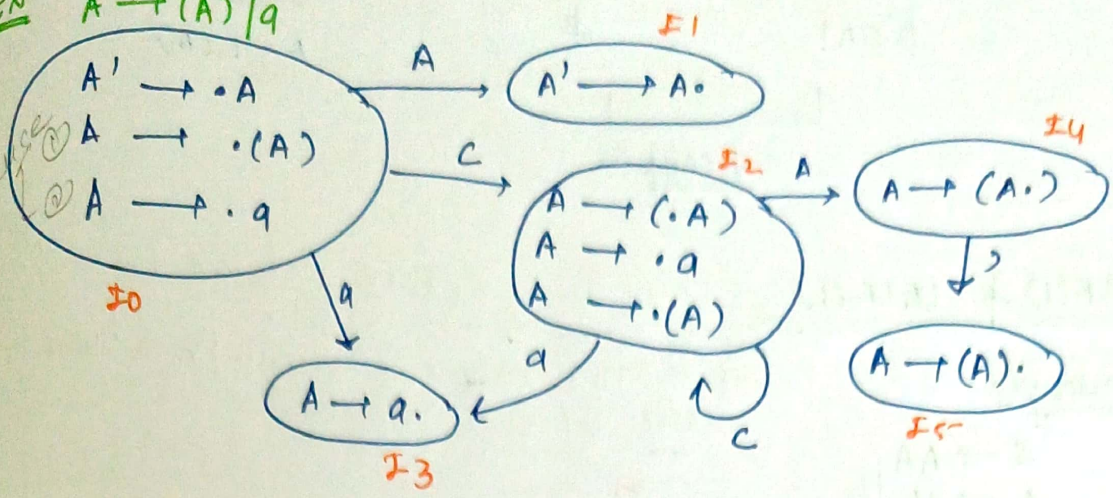
② SLR(1) parsing.

- simple LR
- smallest class of grammer
- few no: of ~~statement~~ states
- simple & fast to construct
- In SLR we place the reduce move only in the follow of left hand side not to entire rows.

- LR(0)
- SRL (1) > LR(0)

- CLR (1)
- LALR (1) > LR(1)

Ex $A \to (A) \mid a$



Parsing table

State	Action				Goto
	a	C	)	$	A
0	S_3	S_2			1
1				Accept	
2	S_3	S_2			4
3			r_2	r_2	
4			S_5		
5			r_1	r_1	

i/p string parsing

Step	parsing stack	i/p	Action
1.	$0	(a)$	shift 2
2.	$0C2	a)$	shift 3
3.	$0C2a3	)$	Reduce $A \to a$
4.	$0C2A4	)$	shift 5
5.	$0C2A4)5	$	

6. $OA1 $

Accept

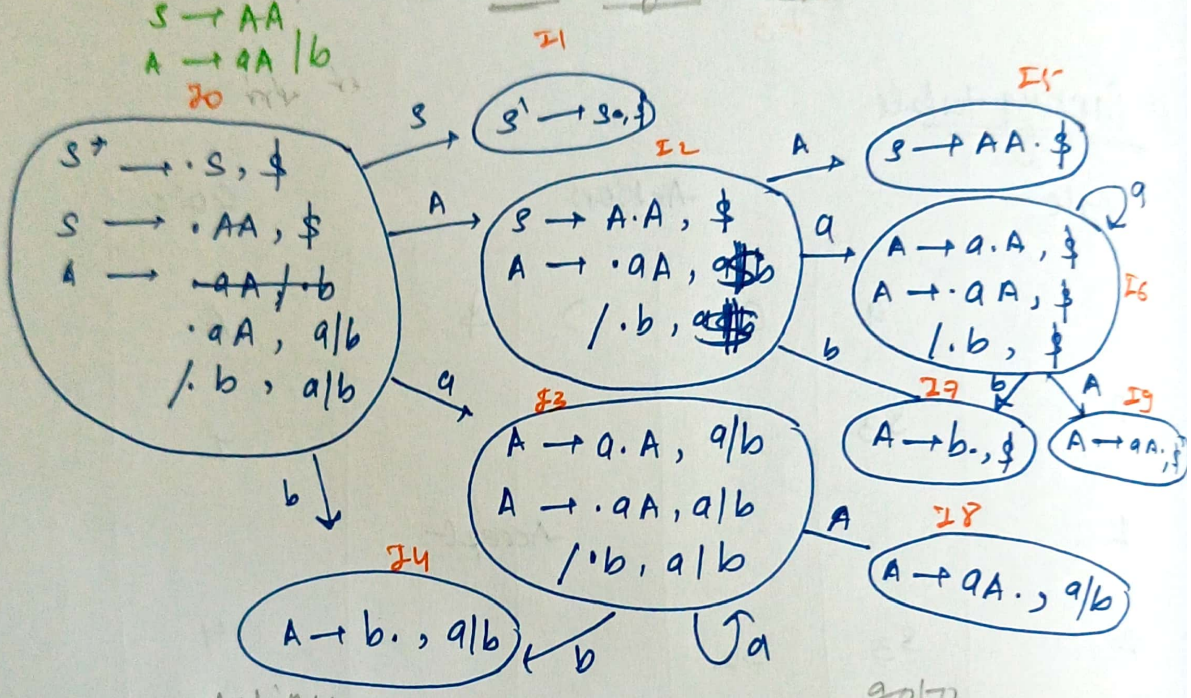
Reduce r_1
$A \to (A)$

(3) CLR(1) & LALR(1)

- To check CLR(1) we have to check that it has conflict that means only one column/row has one or zero input

Example

$S \to AA$
$A \to aA \mid b$



	Action			goto	
	a	b	$	S	A
0	s_4	s_3		1	2
1			accept		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

To check (A)LR(1) now what we will do is
↳ check the same lookahead inputs

I_3, I_6

I_4, I_7

I_8, I_9

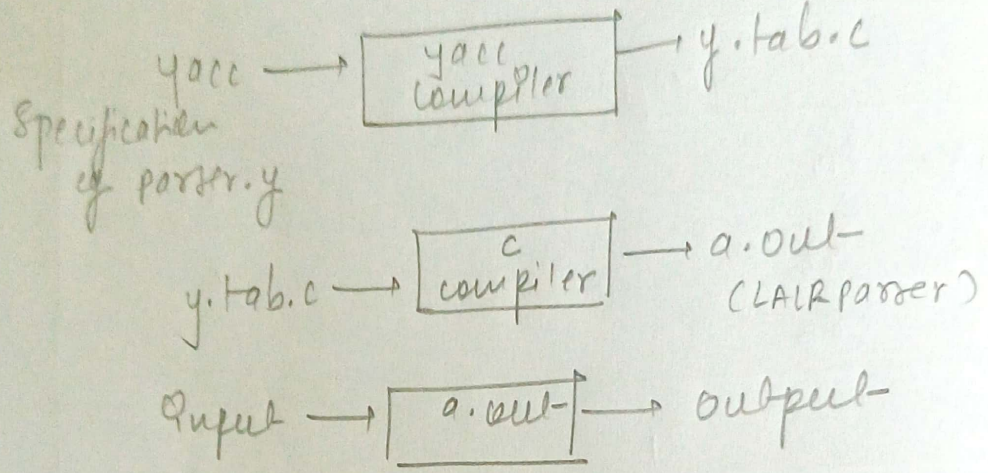
Reduce table of CLR(1) is known as (A)LR(1)

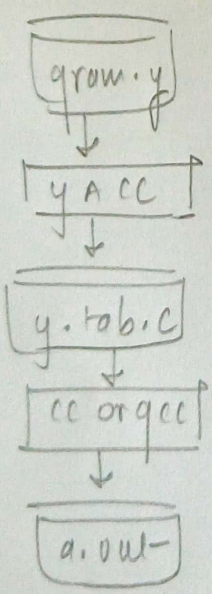
	Action			Goto	
	a	b	$	S	A
0	s_{36}	s_{47}		1	2
1			Accept		
2	s_{69}	s_{74}			5
36	s_{36}	s_{47}			8
47	r_3	r_3			
5			r_1		
6					
7			r_3		
89	r_2	r_2			

- check whether there is conflict
  If conflict then ⊗ LALR(1)
  No conflict ✓ LALR(1)

YACC

- Stands for yet Another compiler compiler
- It is the tool which generate LALR parser.
- Syntax Analyzer (parser) is the second phase of compiler which takes input as token & generate syntax tree.





file containing desired grammar in YACC format

YACC program

C source program created by YACC

compiler

Executable program that will parse grammar given in gram.y.

Syntax

Definitions
%%
Rules
%%
supplementary codes

1) Definition section - All codes betn %{ and %} is copied to the C file. The definition section is where we configure various parser feature such as defining token codes

2) Rules section - Required productions sections is where we specify the grammar rule.

3) Supplementary code section - It is used for ordinary C code that we want copied verbatim to the generated C file.

YACC - It is powerful grammar parser & generator. For eneme, it functions as a tool that takes in a grammar specification & transforms it into executable code capable of meticulously structuring input tokens.

UNIT - 3

Q① Syntax Directed definations?

- It is a kind of abstract specification.
- It is generalised for context free grammar in which each grammar production $X \to a$ is associated with it a set of production rules of the forms $s = f(b_1, b_2 \ldots b_k)$
- Example - $E \to E_1 + T$ { $E.val = E_1.val + T.val$ }
- parse tree uses CFG (context free grammar) to validate the input string & produce output for next phase of the compiler

  Output could be either parse tree or abstract syntax tree. To interleave semantic analysis with the syntax analysis phase of the compiler → we use Syntax Directed translation.

input string → parse tree → Dependency graph → Evaluation order for Semantic Rules.

② A syntax directed definition specifies the values of attributes by associating semantic rules with the grammar productions.

Foreg - An infix → to postfix translator might have a production & rule.

Production	Semantic Rule
$E \to E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

③ A syntax directed translator scheme embeds programs fragments called semantic actions within production bodies

$E \to E_1 + T$ { print '+' }

- Syntax Directed Definition are a key concept in compiler design, particularly in the context of intermediate code generation.
- SDD are used to specify the semantics of a programming language construct by associating semantic rules with the grammar production of a language.

Key Concepts

① Attributes -

- synthesized attributes - whose values are determined from the values of the attributes of the children nodes in a parse tree. (Bottom-up)
- Inherited attributes - whose values are determined from the values of the attributes of the parent or sibling nodes in the parse tree (top-down)

② Grammar production -

- Each production in a grammar is associated with semantic rules that defines how to compute the attributes for that productions
- Ex $E \rightarrow E1 + T$

③ Semantic Rules -

- used to define how the attributes are computed
- this tools are typically written in programming language
- Ex E.code = E1.code || T.code || "add"

④ Intermediate code -

- It is an abstract representation of the source code, often in a form that is easier to optimize & translate into machine code.

Qu① Evaluation order for SDD?

↳ The evaluation order for attributes in SDD's depends on the type of grammar and the dependencies among the attributes.

① S-Attributed Definition

- S-Attribute SDD only use synthesized attributes
- The evaluations order for S-attributed SDD's is straightforward: Evaluate the attributes of its children have been evaluated.
- These can be evalated in a single bottom-up pass over the parse tree.

② L-Attibed Definition

- L-Attributed SDD - allow both inherited and synthesized attributes, but with the restriction that inherited attributes of a node can only depend on the attributes of its ~~parent~~ parent.
- L-attributed SDD - are typically used in top-down parsing. The evaluating order follows the structure of a left-to-right, depth-first traversal of the parse tree.

③ Example: $S \rightarrow AB \{ S.x = f(A.x \mid B.x) \}$ → S-attributed SDD

EX: $S \rightarrow AB \{ A.x = S.x + 2 \}$ → L-Attributed SDD

④ Dependency graph -

- To determine a valid evaluation order for a general SDD, a dependcy graph can be constructed.
- Nodes in the graph represent attributes and edges represent dependencies between them
- If the dependency graph is acyclic, the attributes can be evaluates in an order consistents with the dependencies.
- If the graph has cycles, the SDD cannot be evaluated in a well-defined order without additional mechanisms.

Ques Explain semantic Analysis?

- It is the third phase of compiler.
- It make sure that declaration & statement of program are semantically correct
- It is a collection of procedure which is called by parser as and when required by grammar.
- type checking is an important part of semantic Analysis where compiler make sure that each operator has matching operands.

Semantic Analyzer

- It uses syntax tree & symbol table to check whether the given program is semantically consistent with language def.
- It gather type info & stores it in either syntax tree or symbol table.

Semantic Error

- type mismatch
- undeclared variables
- Reserved identifier misuse.

Function of semantic Analysis:

1. type checking – Ensure that datatype are used in a way consistent with their def.
2. Label checking – A program should contain labels references
3. Flow control check – keeps a check that control structures are used in a proper manner.