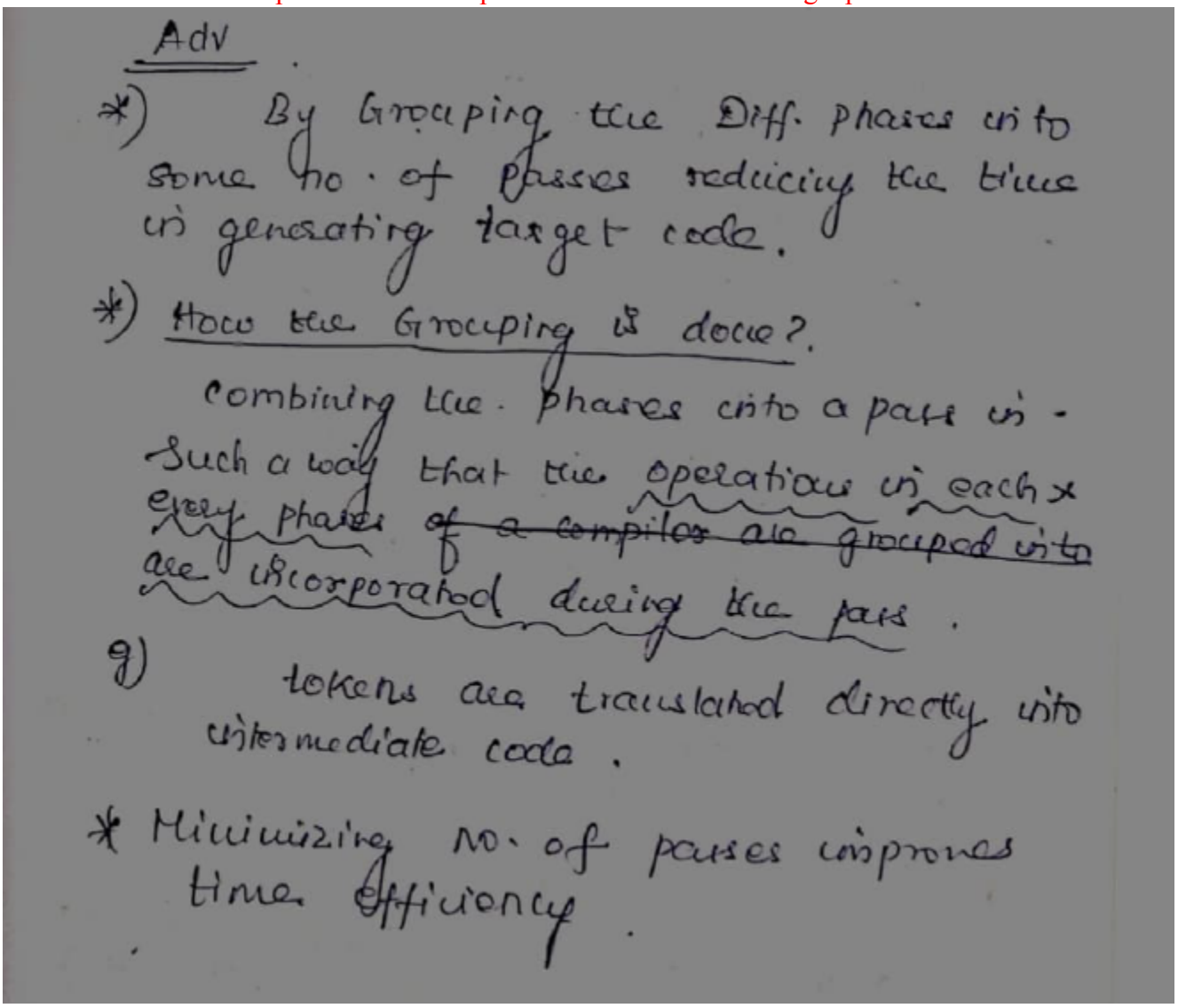
Sona College of Technology

Department of CSE

Compiler Design CIE questions

Would it be better if the phases of the compiler are combined into a single phase?



Adv

*) By Grouping the Diff. phases into some no. of passes reducing the time in generating target code.

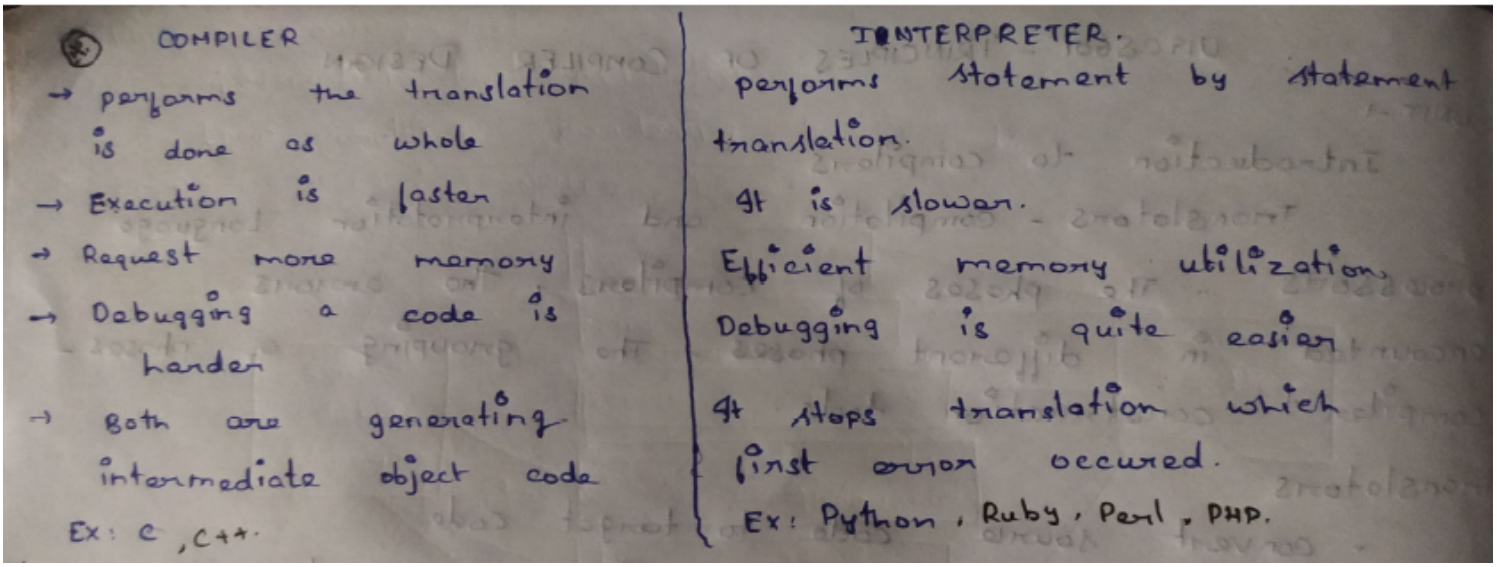
*) How the Grouping is done?

Combining the phases into a pass in such a way that the operations in each & every phase of a compiler are grouped into one incorporated during the pass.

g) tokens are translated directly into intermediate code.

* Minimizing no. of passes improves time efficiency.

Differentiate compilers from interpreters.



COMPILER

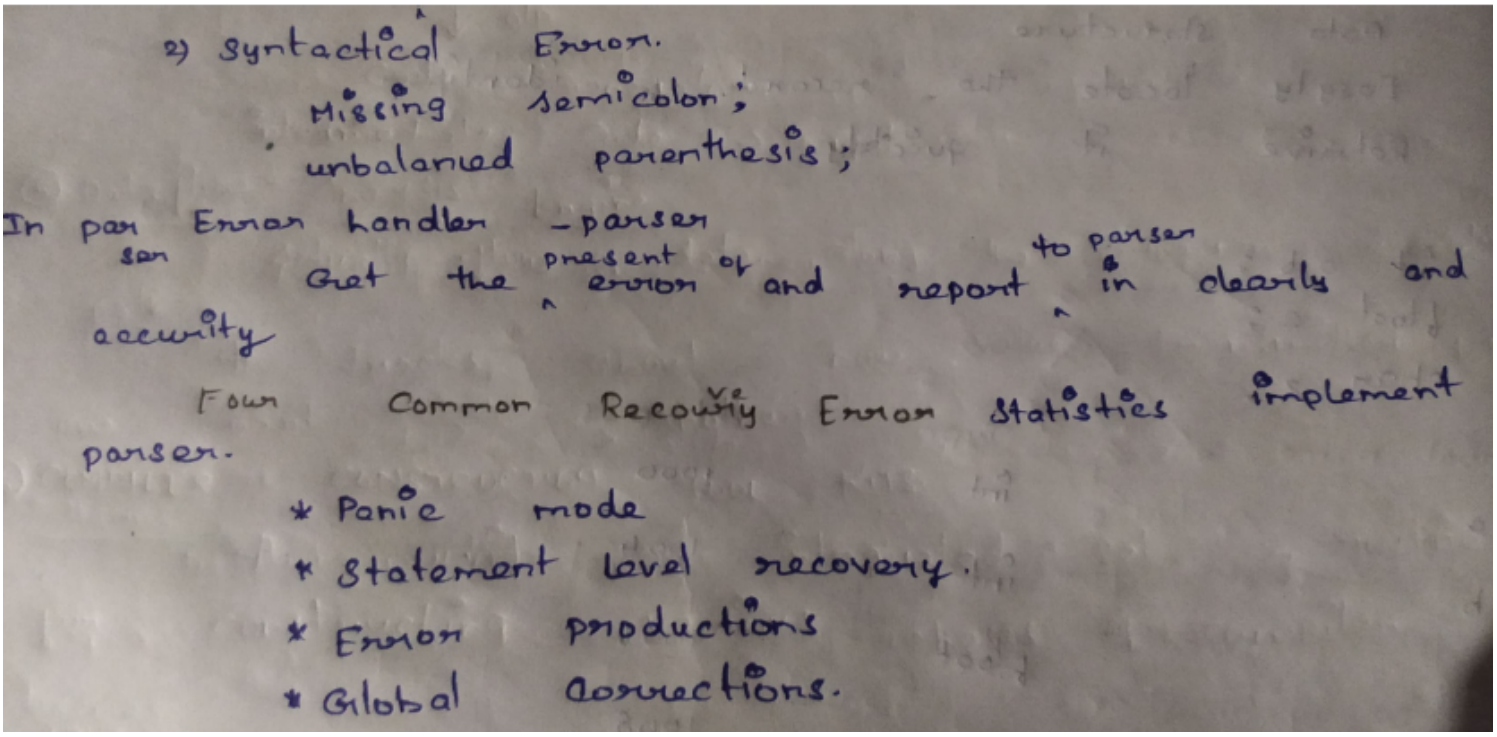
→ performs the translation is done as whole
→ Execution is faster
→ Request more memory
→ Debugging a code is harder
→ Both are generating intermediate object code

Ex: C, C++.

INTERPRETER.

performs statement by statement translation.
It is slower.
Efficient memory utilization.
Debugging is quite easier
It stops translation which first error occured.
Ex: Python, Ruby, Perl, PHP.

An arithmetic expression with unbalanced parenthesis is lexical or syntax error. Comment on it.



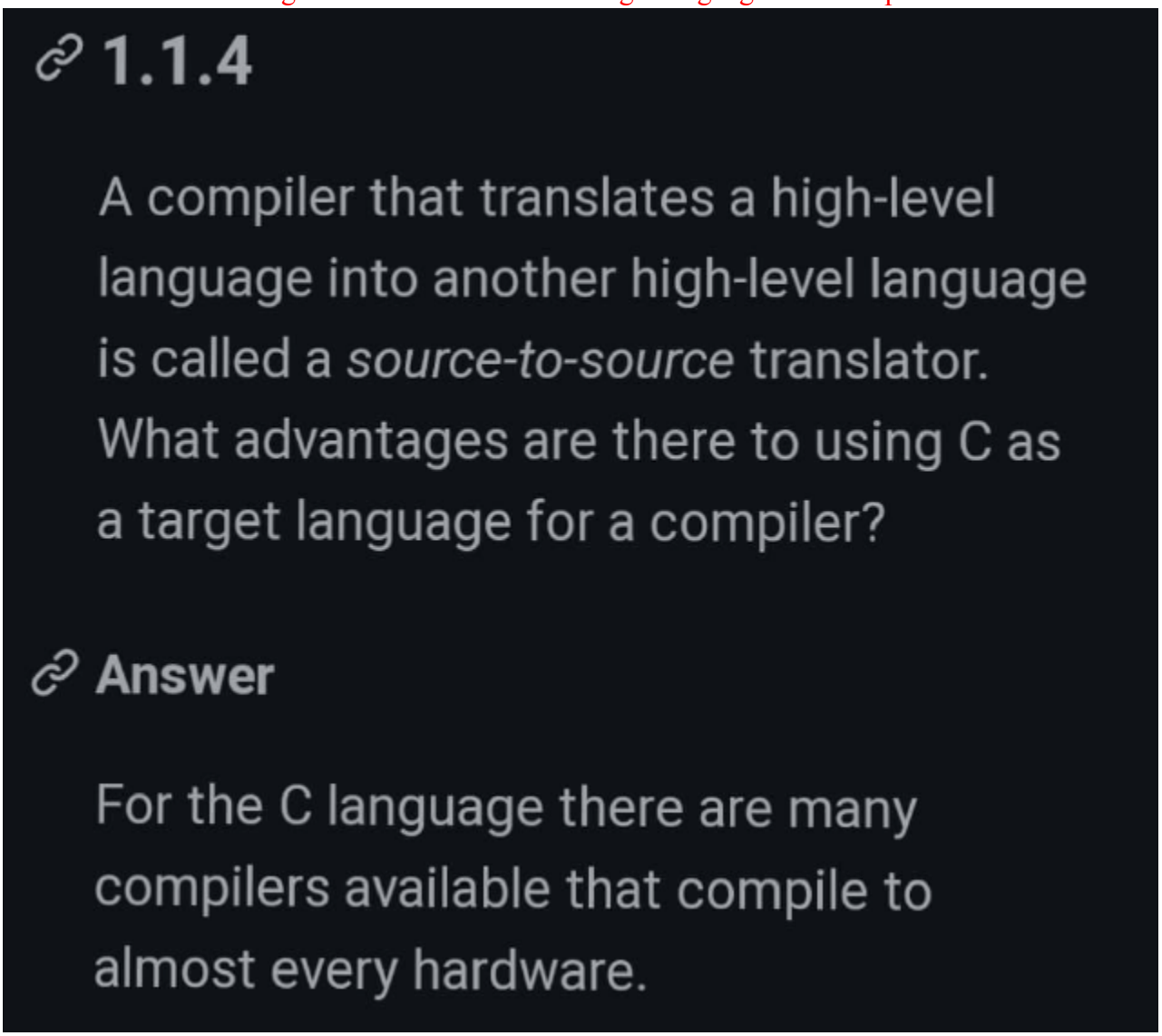
2) Syntactical Error.
Missing semicolon;
unbalanced parenthesis;

In parser Error handler - parser present of error and report to parser in clearly and accurity

Four Common Recovery Error Statistics implement parser.

* Panic mode
* Statement level recovery.
* Error productions
* Global corrections.

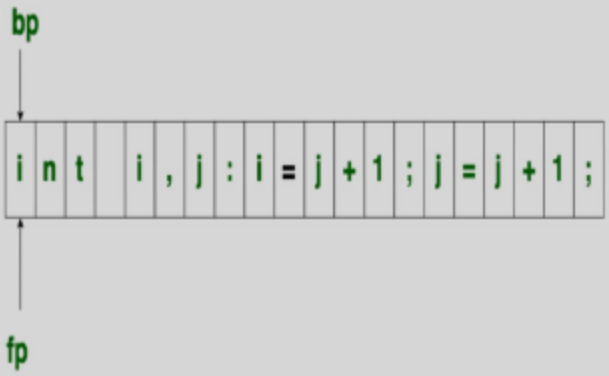
A compiler that translates a high-level language into another high-level language is called a source- to-source translator. What advantages are there to use C as a target language for a compiler?



Point out why is buffering used in lexical analysis? What are the commonly used buffering methods?

Input Buffering in Compiler Design

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(bp) and forward to keep track of the pointer of the input scanned.



Initial Configuration

TYPES

1.One buffer scheme
2.Two buffer scheme

Obtain the regular expressions for the following sets:

a) The set of all strings over {a, b} beginning and ending with 'a'.
b) The set of all strings over {b} such that string belong to $\{b^2, b^5, b^8, \ldots\}$.

(3) (a) the set of all strings over {a,b} beginning and ending with 'a'

a (a|b)* a

(b) the set of all strings over {b} such that string belongs to {b2, b5, b8 ... }

bb (bbb)*

Comment on the efficiency of the compiler if the number of passes in compilation is increased

Adv

*) By Grouping the Diff. phases into some no. of passes reducing the time in generating target code.

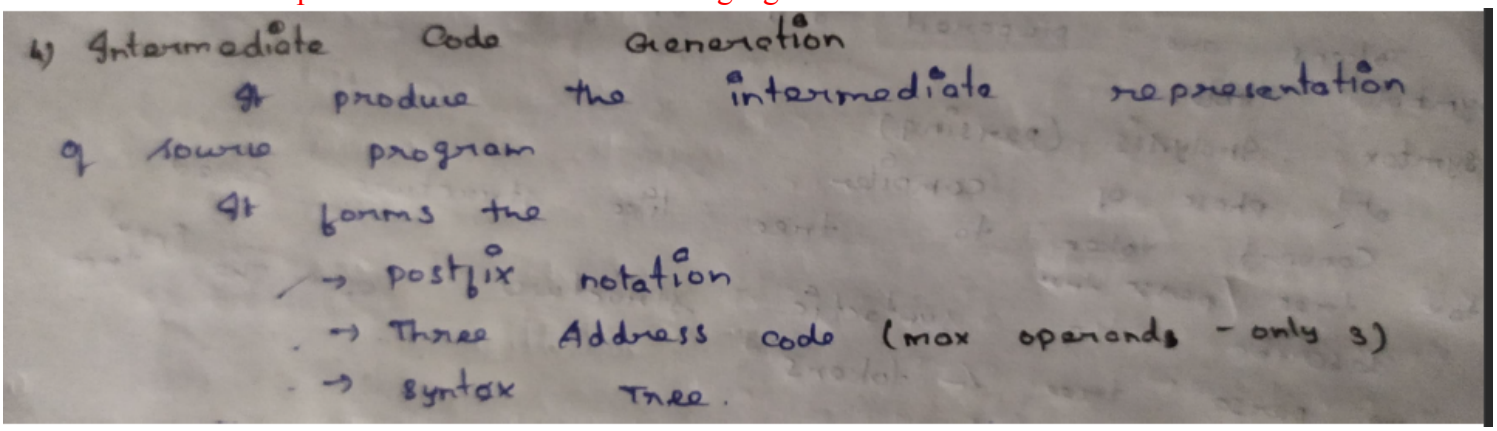
*) How the Grouping is done?

Combining the phases into a pass in such a way that the operations in each x grouped into are incorporated during the pass.

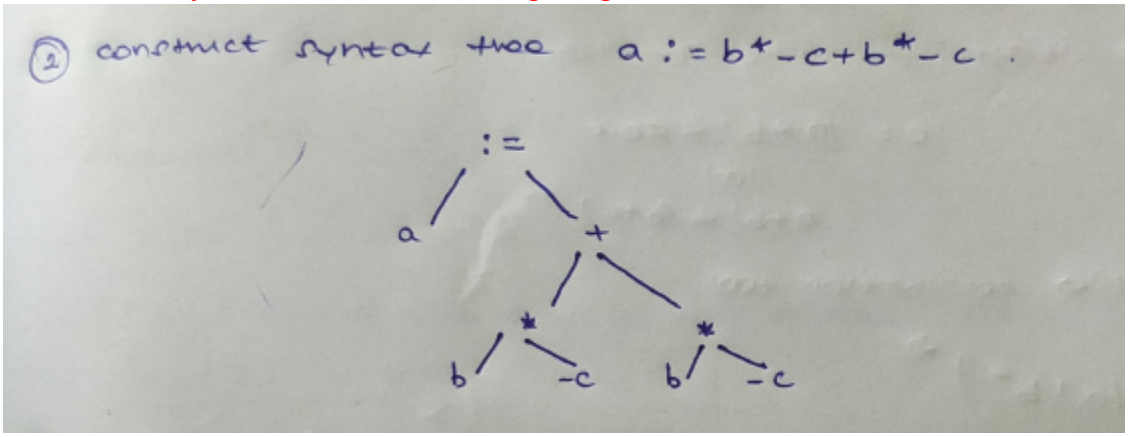
g) tokens are translated directly into intermediate code.

* Minimizing no. of passes improves time efficiency.

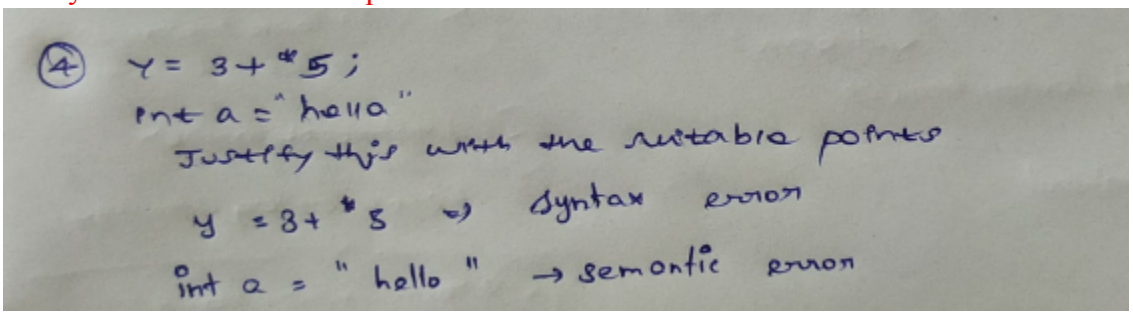
What are the various representations of Intermediate languages?



Construct the syntax tree for the following assignment statement: a:=b*- c+b*-c.



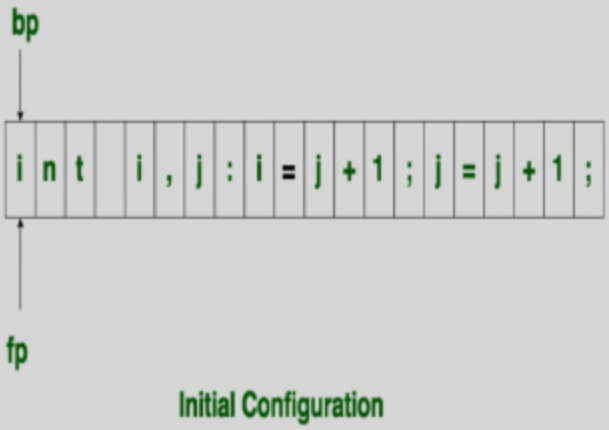
Identify the type of error.
y = 3 + * 5;
int a = "hello";
Justify this with the suitable points..



Point out why is buffering used in lexical analysis? What are the commonly used buffering methods?

Input Buffering in Compiler Design

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers begin ptr(bp) and forward to keep track of the pointer of the input scanned.



TYPES

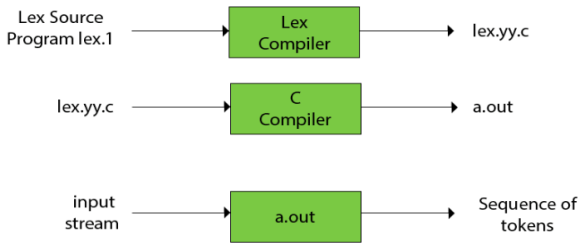
1.One buffer scheme
2.Two buffer scheme

Describe the language denoted by the regular expressions a*ba*ba*ba*.

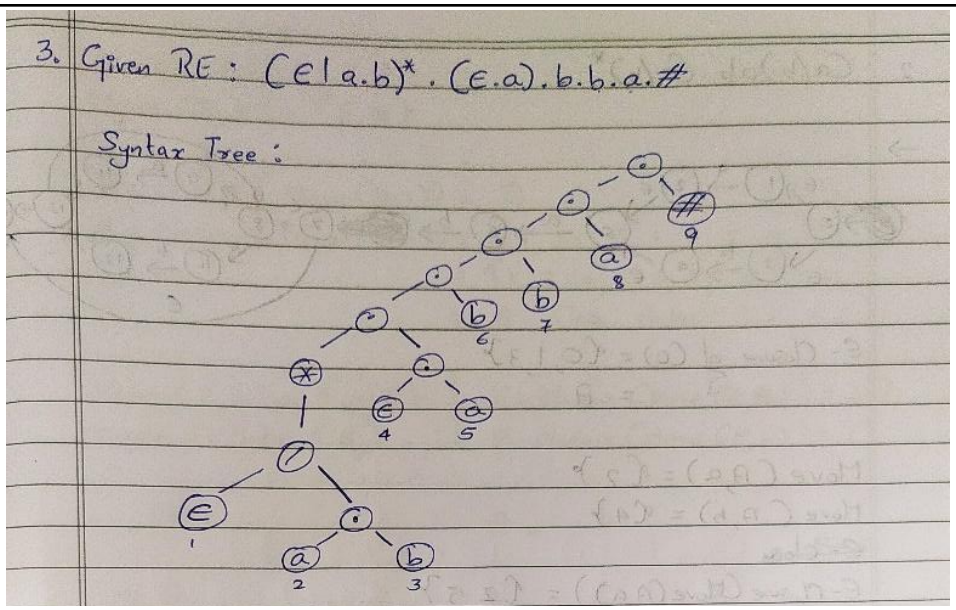
How can a lexical analyzer be constructed using Lex tool?

The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Draw the syntax tree for the augmented regular expression (ε | ab)*(εa)bba#



Mention the basic issues in parsing.

Issues in Parsing:

- Variable redeclaration
- Variable initialization before use
- Datatype mismatch for an operation
- Specification of syntax
- Representation of input after parsing.

What is the role of the parser in a compiler model?

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It must report any syntax errors if occurs.

Consider the grammar g=(V,T,P,S). Here V={S,N,N_p,ADJ} and T={and, eggs, ham, pencilgreen, cold, tasty}. The set contains the following rules:

Np Np and Np N eggs | ham | pencil|

Np ADJ Np ADJ green |cold |tasty|

Np N

Show that the grammar is ambiguous by constructing two different parse trees for the sentence “green eggs and ham”.

2.2 Ambiguity

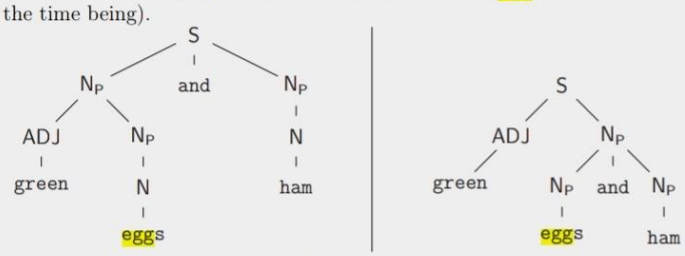
Consider the following grammar $\mathcal{G} = (V, \Sigma, R, \mathsf{S})$. Here

$$V = \{\mathsf{S}, N, \mathsf{N_P}, \mathsf{ADJ}\} \qquad \text{and} \qquad \Sigma = \{\texttt{and}, \texttt{eggs}, \texttt{ham}, \texttt{pencilgreen}, \texttt{cold}, \texttt{tasty}, \ldots\}.$$

The set R contains the following rules:

- $\mathsf{N_P} \to \mathsf{N_P}$ and $\mathsf{N_P}$
- $\mathsf{N_P} \to \mathsf{ADJ}\ \mathsf{N_P}$
- $\mathsf{N_P} \to \mathsf{N}$
- $\mathsf{N} \to$ eggs | ham | pencil | ...
- $\mathsf{ADJ} \to$ green | cold | tasty | ...
- ...

Here are two possible parse trees for the string green eggs and ham (ignore the spacing for the time being).



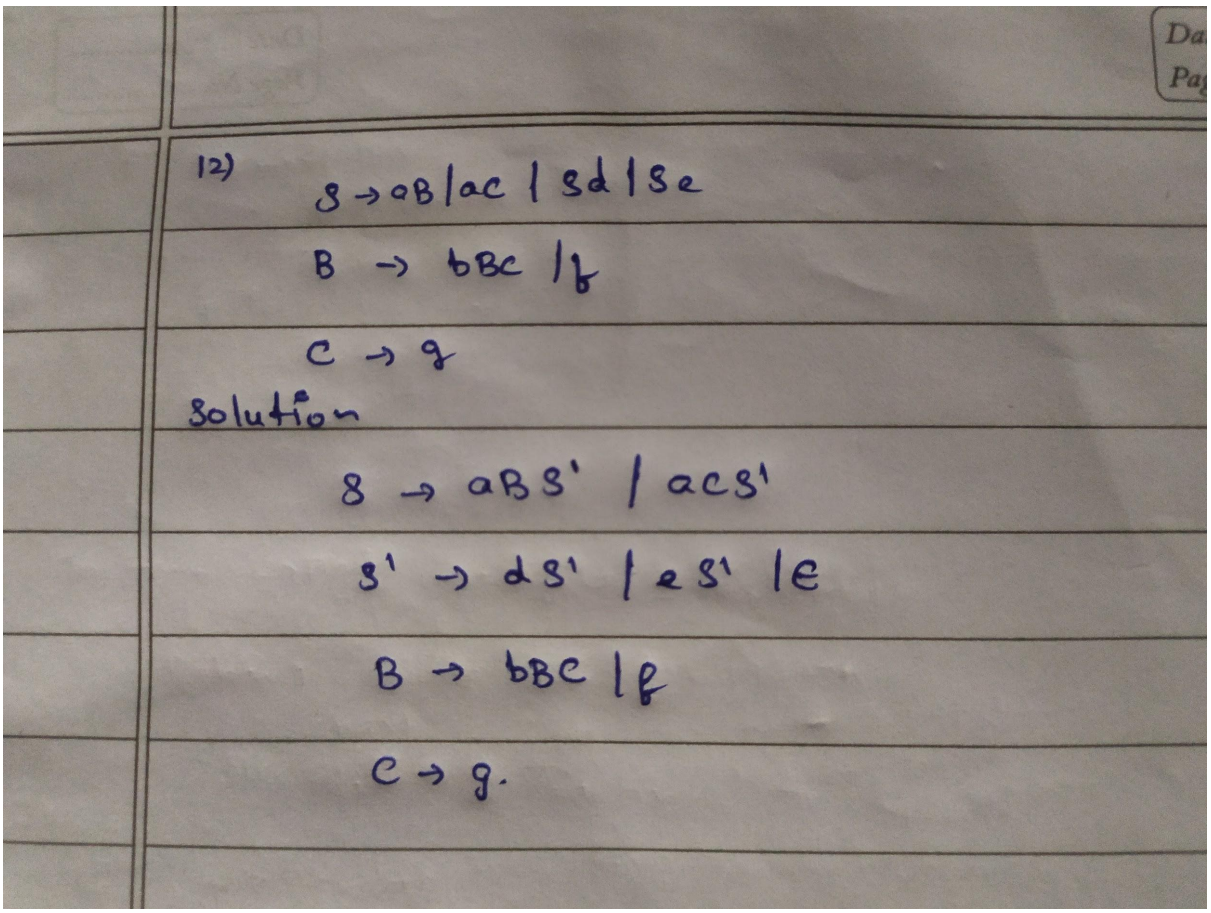
The two parse trees group the words differently, creating a different meaning. In the first case, only the eggs are green. In the second, both the eggs and the ham are green.

A string w is ***ambiguous*** with respect to a grammar $\mathcal{G}$ if w has more than one possible parse tree using the rules in $\mathcal{G}$.

Most grammars for practical applications are ambiguous. This is a source of real practical issues, because the end users of parsers (e.g. the compiler) need to be clear on which meaning is intended.

Eliminate Left recursion for the following grammar.

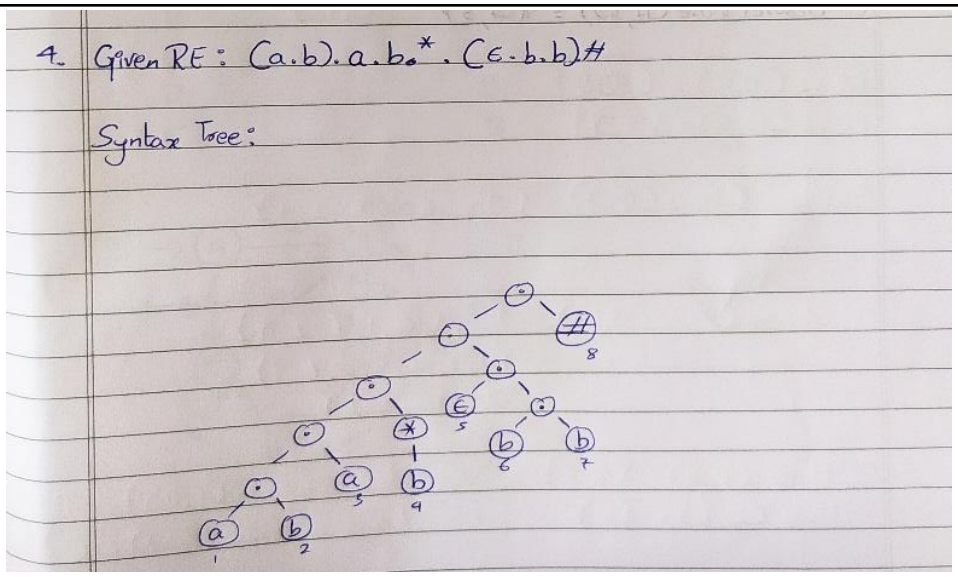
S → aB | aC | Sd | Se
B → bBc | f
C → g



Write short notes on LEX.

Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Draw the syntax tree for the augmented expression (ab)ab*(ε bb)#



Write a context free grammar that generates the set of all strings of a's and b's which is a palindrome.

Let P be language of palindromes with alphabet {a, b}. One can determine a CFG for P by finding a recursive decomposition.

CFG is:

$P \rightarrow aPa \mid bPb \mid a \mid b \mid \varepsilon$

How will you eliminate left factor in a grammar?

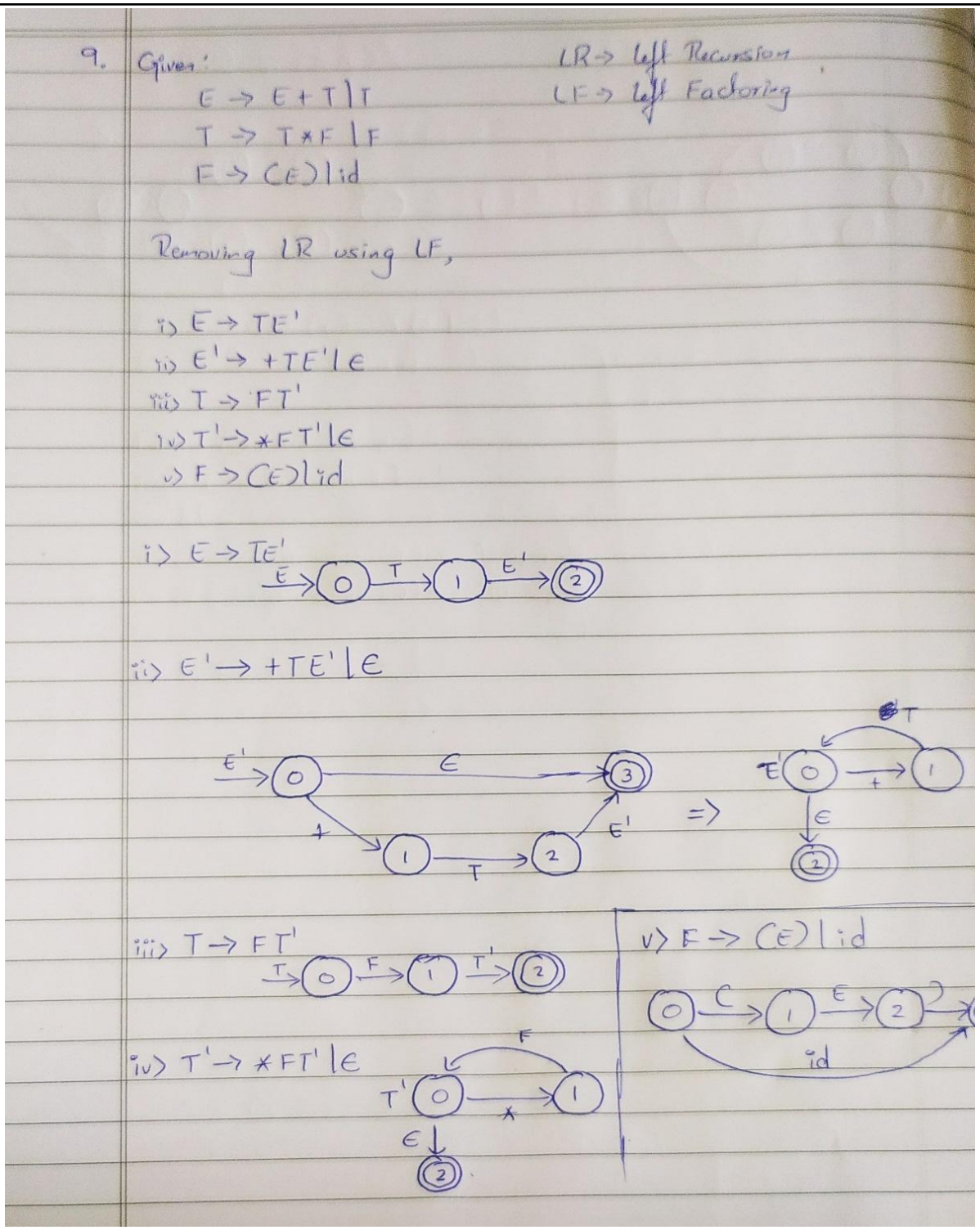
A Grammar G (V, T, P, S) is left recursive if it has a production in the form.
$A \rightarrow A\,\alpha \mid \beta$.
The above Grammar is left recursive because the left of production is occurring at a first position on the right side of production. It can eliminate left recursion by replacing a pair of production with
$A \rightarrow \beta A'$
$A \rightarrow \alpha A' \mid \epsilon$

Draw transition diagrams for predictive parsers for the grammar

E E + T | T

T T * F | F

F (E) | id



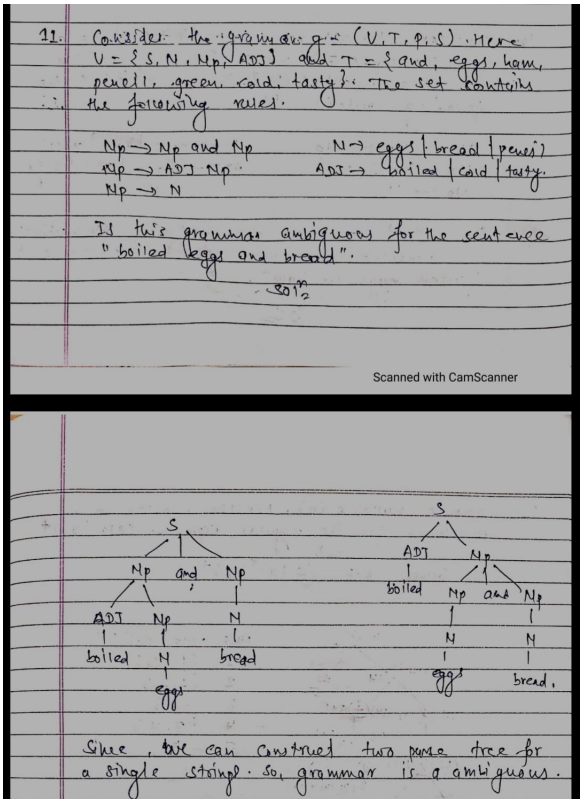
Consider the grammar g=(V,T,P,S). Here V={S,N,N_p,ADJ} and T={and, eggs, ham, pencil green, cold, tasty}. The set contains the following rules:

Np Np and Np N eggs | bread| pencil|

Np ADJ Np ADJ boiled |cold |tasty|

Np N

Is this grammar ambiguous for the sentence “boiled eggs and bread”.

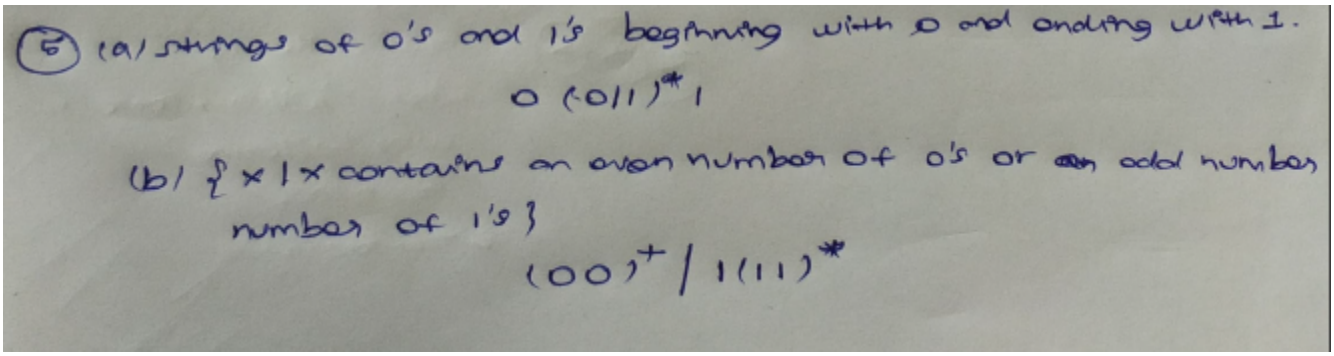


What is a front end and back end of a compiler? What are the advantages of breaking up the compiler functionality into these two distinct stages?

Obtain the regular expressions for the following having Σ={0,1}

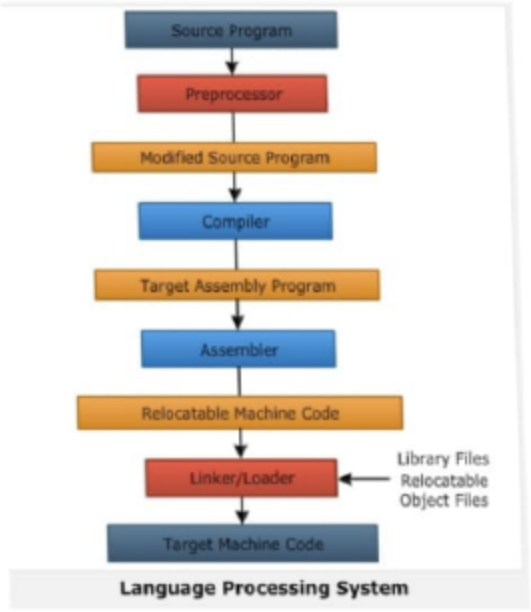
a) strings of 0’s and 1’s beginning with 0 and ending with 1.

b) {x | x contains an even number of 0’s or an odd number of 1’s}



Depict diagrammatically and explain how a language is processed?

Language Processing System



Language Processing System

Language Processing System.

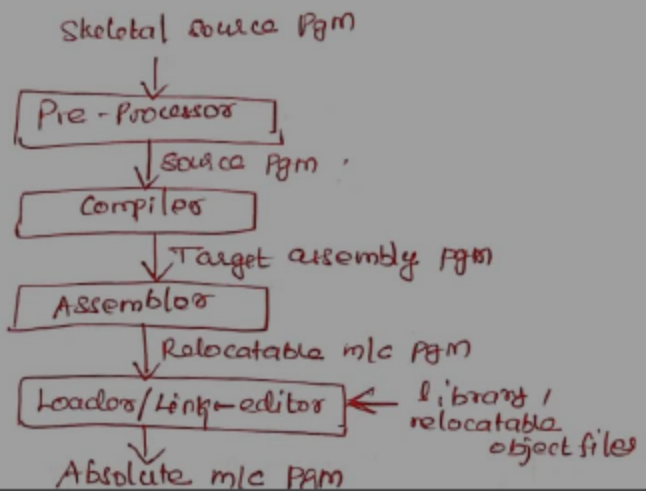
ProProcessors

> A Source pgm may be divided into modules stored in separate file. The task of collecting ~~macros~~ the source pgm is entrusted to a separate pgm called preprocessor.

> It also expand macros into source language statement

Compilers

compiler is a pgm that takes source pgm as input & produces assembly language pgm as output.



Assembler

It is a pgm that converts assembly language pgm into m/c lang. pgm. It produces re-locatable m/c code as its output

Loader and Linker editor

> The relocatable m/c code has to be linked with other re-locatable object files & library files in to the code that actually runs on the machine

> The linker resolves external memory addresses, where the code in one file may refer to a location in another file

> The loader puts together the entire executable object files into memory for execution

Preprocessor:
It is a program that reads the source code and prepares it for the translator
The result of preprocessing is called the compiler

Roles of Preprocessor:
Copies all library functions of a file into a source code
Expand macros into a source code

Example: #include<stdio.h> here # is a processor directive. It copies all library functions of stdio.h (standard input output) file into a program.

Compiler:
Compiler is a program that reads a program in high level language known as source code as its input and converts it into an equivalent program in low level language known as object code as its output.

Assembler

The compiler may produce an assembly language is easier to produce as output and is easier to debug .The assembly language is then processed by a program called assembler that produce relocatable machine code as its output.

- Relocatable files are not human readable format

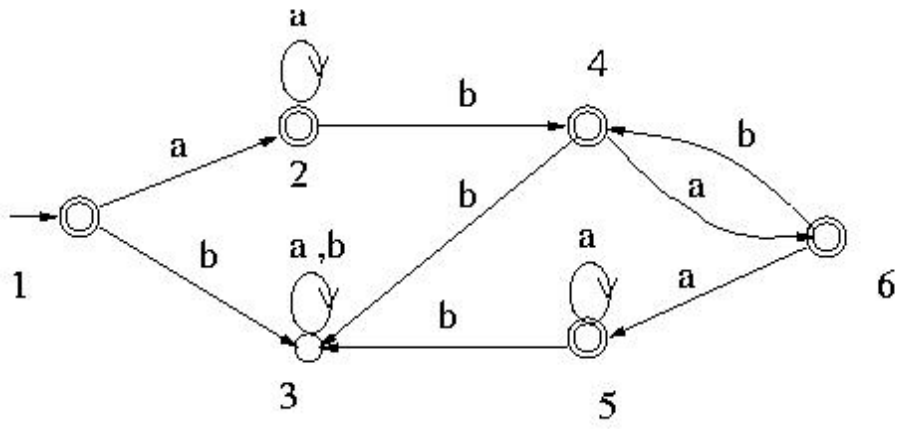
Linker/ Loader

The process of linking user functions and system functions to a final executable code known as linker

Loader:
Once the program is linked, it is ready for execution. The process of loading the program into primary memory for execution is known as loader

- The loader finds sufficient place for the program to be loaded for execution.

minimize the number of states of the following DFA.



Transition Table

states	a	b
→ * 1	2	3
* 2	2	4
3	3	3
* 4	6	3
* 5	5	3
* 6	5	4

$\pi_0 = \{ Q_1^0, Q_2^0 \}$

where Q_1^0 is set of all final states

$Q_2^0 = Q - Q_1^0$, Q is a set of all the states in DFA

$Q_1^0 = \{ 1, 2, 4, 5, 6 \}$

$Q_2^0 = \{ 3 \}$

$\pi_0 = \{ \underbrace{\{1, 2, 4, 5, 6\}}_{1}, \underbrace{\{3\}}_{2} \}$

	1	2	4	5	6
a	1	1	1	1	1
b	2	1	2	2	1

$\therefore \pi_1 = \{ \underbrace{\{3\}}_{2}, \underbrace{\{1, 4, 5\}}_{3}, \underbrace{\{2, 6\}}_{4} \}$

Again;

	1	2	4	5	6
a	4	4	4	3	3
b	2	3	2	2	3

	1	2	4	5	6
a	4	4	4	3	3
b	2	3	2	2	3

$\therefore \pi_2 = \{ \underset{2}{\{3\}}, \underset{5}{\{1,4\}}, \underset{6}{\{2\}}, \underset{7}{\{5\}}, \underset{8}{\{6\}} \}$

	1	2	4	5	6
a	6	6	8	7	7
b	2	5	2	2	5

$\therefore \pi_3 = \{ \underset{2}{\{3\}}, \underset{9}{\{1\}}, \underset{6}{\{2\}}, \underset{10}{\{4\}}, \underset{7}{\{5\}}, \underset{8}{\{6\}} \}$

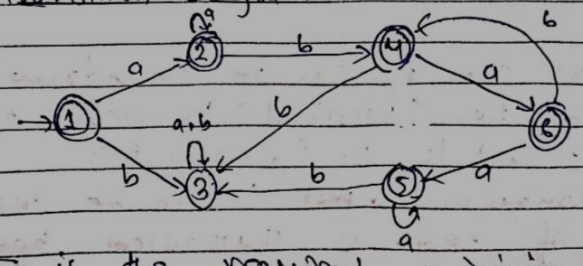
	1	2	4	5	6
a	6	6	8	7	7
b	2	10	2	2	10

$\therefore \pi_4 = \{ \{3\}, \{1\}, \{2\}, \{4\}, \{5\}, \{6\} \}$

Transition Table

states	input a	input b
→ * 1	2	3
* 2	2	4
3	3	3
* 4	6	3
* 5	5	3
* 6	5	4

Transition Diagram:



This is the required minimized DFA.

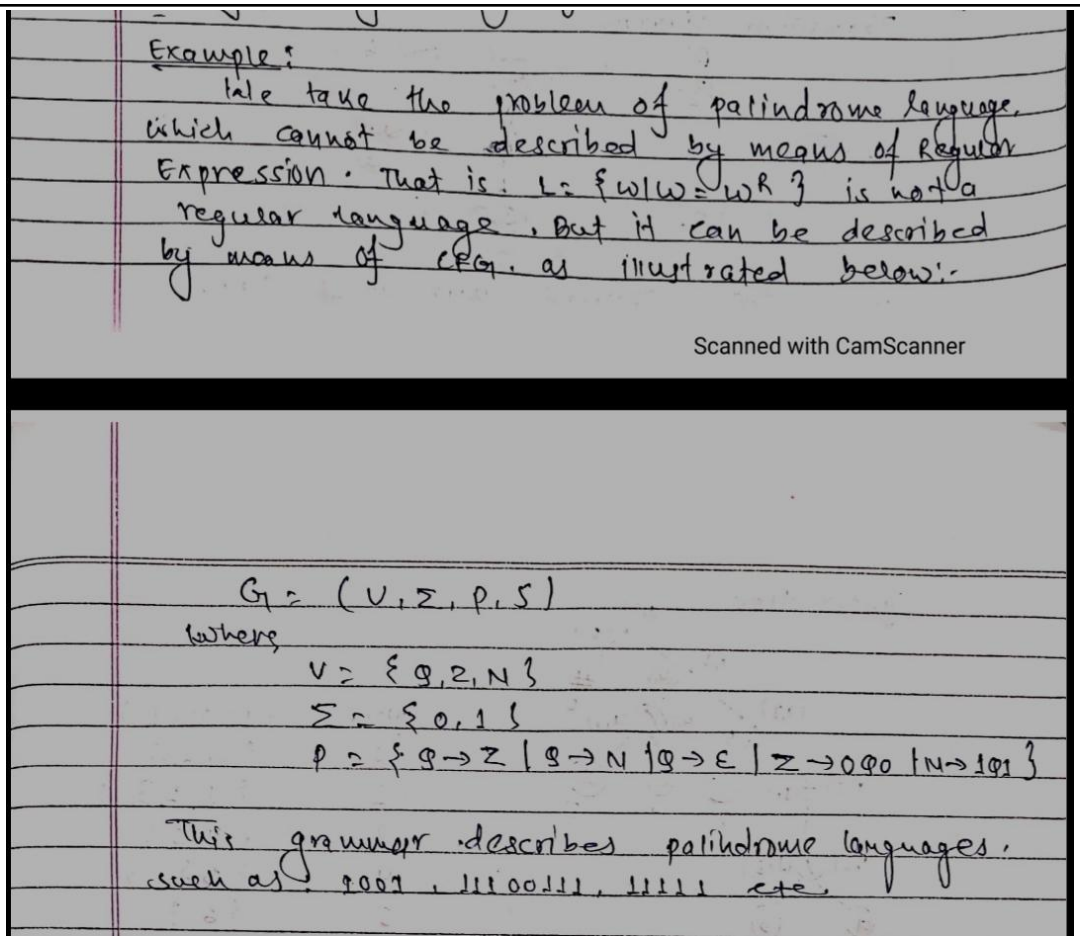
To verify the syntax of programming language constructs, compilers use CFG.Why not Regular expression?. Justify with examples.

Regular Expressions are capable of describing the syntax of Tokens. Any syntactic construct that can be described by Regular Expression can also be described by the Context free grammar.

grammar that's why we are using Regular Expression. There are several reasons and they are:

Regular Expressions	Context-free grammar
Lexical rules are quite simple in case of Regular Expressions.	Lexical rules are difficult in case of Context free grammar.
Notations in regular expressions are easy to understand.	Notations in Context free grammar are quite complex.
A set of string is defined in case of Regular Expressions.	In Context free grammar the language is defined by the collection of productions.
It is easy to construct efficient recognizer from Regular Expressions.	By using the context free grammar, it is very difficult to construct the recognizer.
There is proper procedure for lexical and syntactical analysis in case of Regular Expressions.	There is no specific guideline for lexical and syntactic analysis in case of Context free grammar.
Regular Expressions are most useful for describing the structure of lexical construct such as identifiers, constant etc.	Context free grammars are most useful in describing the nested chain structure or syntactic structure such as balanced parenthesis, if else etc. and these can't be define by Regular Expression.

Article Tags : Compiler Design GATE CS



Example:

Let take the problem of palindrome language, which cannot be described by means of Regular Expression. That is. $L = \{w | w = w^R\}$ is not a regular language. But it can be described by means of CFG. as illustrated below:-

Scanned with CamScanner

$G = (V, \Sigma, P, S)$

where,

$V = \{S, Z, N\}$

$\Sigma = \{0, 1\}$

$P = \{S \to Z \mid S \to N \mid S \to \varepsilon \mid Z \to 0S0 \mid N \to 1S1\}$

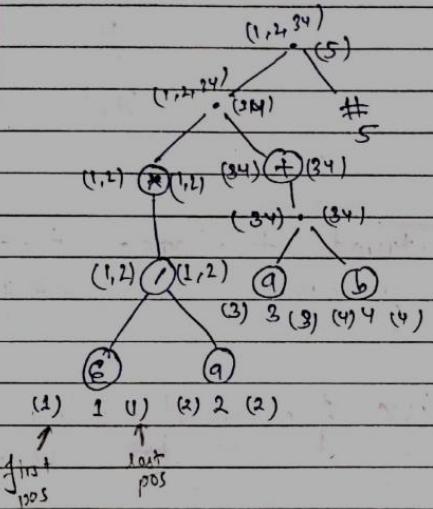
This grammar describes palindrome languages, such as: 1001, 11100111, 11111 etc.

Compute nullable(n), FIRSTPOS, LASTPOS and FOLLOWPOS for the regular expression
(ε | a)*(ab)+

13. Compute nullable (n), FIRSTPOS, LASTPOS and FOLLOWPOS for the regular expression $(\epsilon|a)^*(ab)^+$.

Soln:

Given, $R.E = (\epsilon|a)^*(ab)^+$



Node	Follow pos (i)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{3,4}
5	ϕ

Write the rules for computing the FIRST and FOLLOW functions for constructing predictive parsing table.

Computation of First

Rules

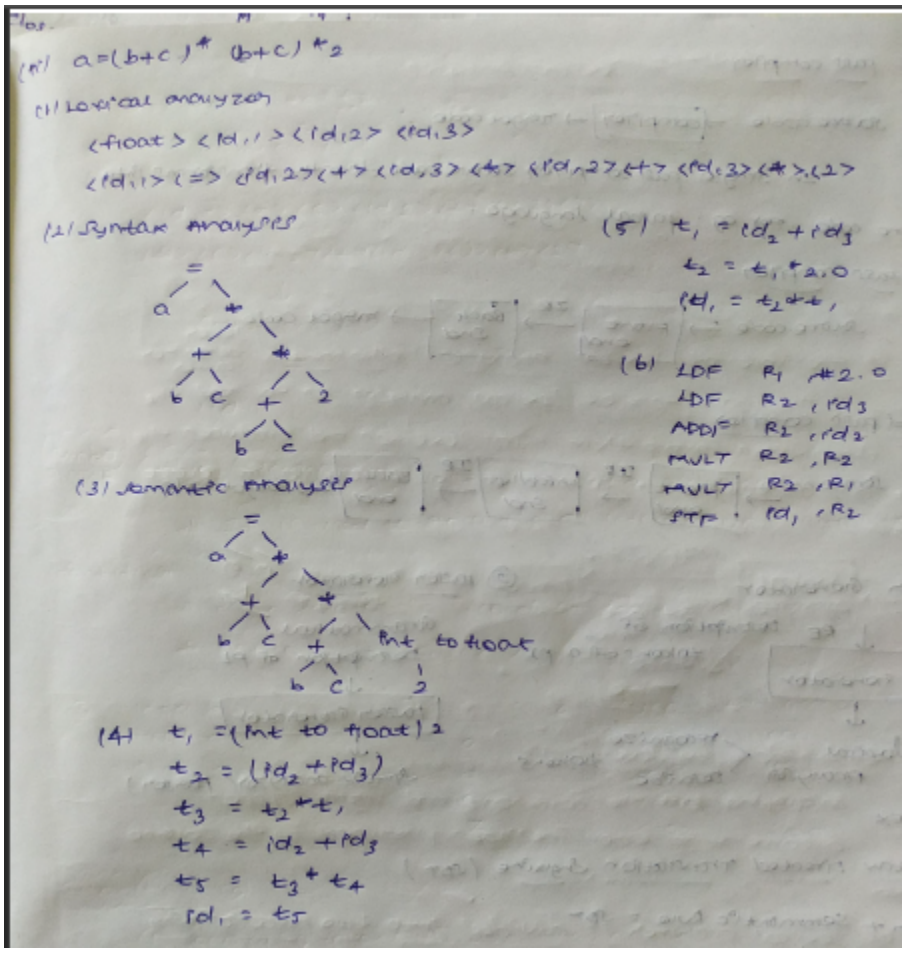
1) If x is a terminal, then First(x) = {x}
2) If $x \to \epsilon$ is a production, then add ϵ to First(x).
3) If x is a non terminal and $x \to Y, Y_1 Y_2 \dots Y_k$ is a production, then add FIRST(Y_1) to FIRST(x) If Y_1 derives ϵ, then add FIRST(Y_2) to FIRST(x)

COMPUTATION OF FOLLOW

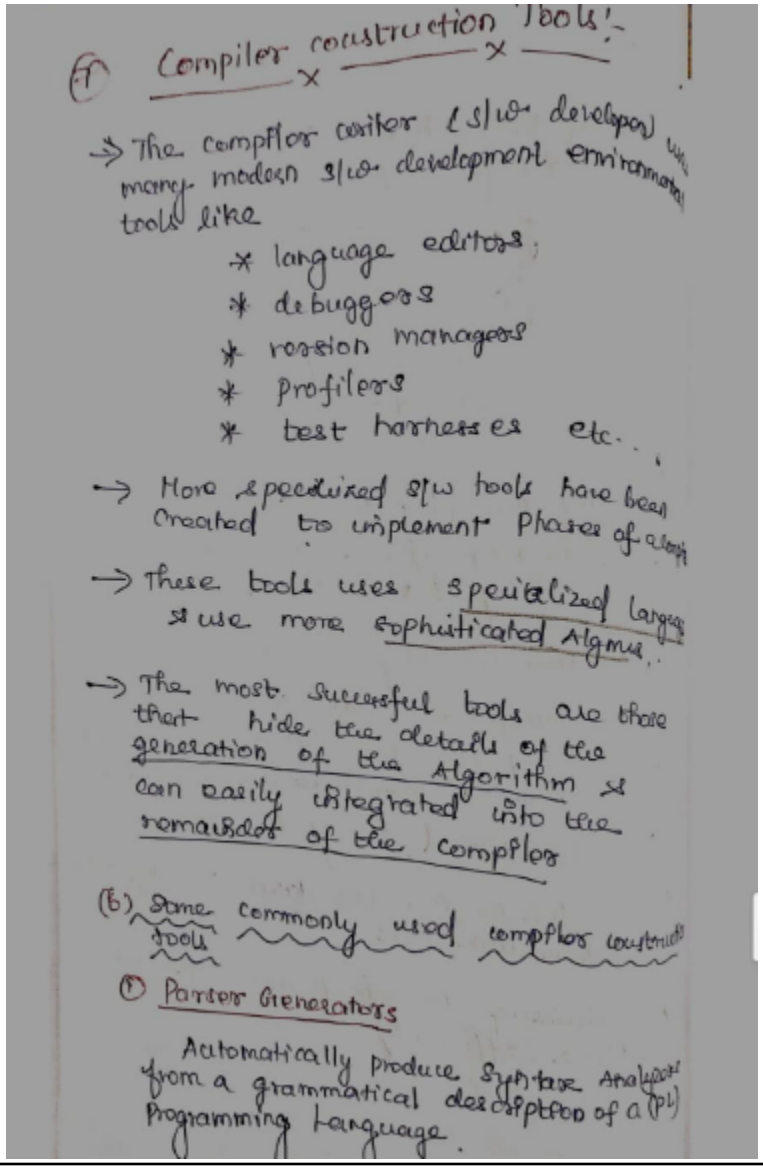
Rules.

1) For the follow (start symbol) place $ where $ is the input end marker.
2) For there is a production $A \to \alpha B\beta$, then everything in FIRST(β) except ϵ in FOLLOW(B)
3) For there is a production $A \to \alpha B$ or a production $A \to \alpha B\beta$ where FIRST(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B)

(i)

(ii)	Generate the target code for the following code fragment. Assume a=(b+c) * (b+c) * as input to the lexical analyzer and try to show the intermediate outputs of all the phases from lexical analyzer to code generator. 
How do you speed up the reading the source program? What are the specialized algorithms used by the compiler? Did all of them read the source program with same time or they differ? Justify with suitable claims and figures.	
Can you identify the lexemes that make up the tokens in the following program segment? Write the corresponding tokens, lexemes and pattern. **void swap(int i, int j)** **{ int t; t=i; i=j; j=t;**	
Do compiled languages beat interpreted languages in performance? What is the difference between those languages? Justify this with appropriate examples.	
Construct minimized Deterministic Finite Automata to accept the regular expression : (0+1)* (00+11) (0+1)*.	
Illustrate the output of each phase of compilation while translating the following assignment statement i = i*70+j+2;.	
How to solve the source program to target machine code by using language processing system?	
Can you identify and list the systems/tools to help with the compiler-writing process are often been referred to as compiler-compilers, compiler-generators or translator-writing systems? Why are they called	

compiler-compilers? Write the representation and implementation techniques? What flexibility do they provide?



(a) Compiler construction Tools:-

→ The compiler writer (s/w developer) uses many modern s/w development environmental tools like

* language editors
* debuggers
* version managers
* profilers
* test harnesses etc.

→ More specialized s/w tools have been created to implement phases of a compiler

→ These tools uses specialized languages & use more sophisticated algorithms.

→ The most successful tools are those that hide the details of the generation of the algorithm & can easily integrated into the remainder of the compiler

(b) Some commonly used compiler construction tools

① Parser Generators

Automatically produce syntax analyzers from a grammatical description of a (PL) Programming Language.

③ Scanner Generators

Generates lexical Analyzer from a regular expression description of the tokens of a language

④ Syntax-directed translation engines

[Parse tree → (to) → Intermediate code]

→ Produce collections of routines for walking a parse tree & generating intermediate code

⑤ Code Generators (Generator)

↑

Produce a code generator from a collection of rules for translating each operation of intermediate lang. into the machine lang. for a target machine.

⑥ Data flow Analysis Engines

→ It facilitates the gathering of information about how values are transmitted from one part of a pgm to each other part.

→ It is a key part of code optimization

⑦ Compiler construction tool kits

→ Provide an integrated set of routines for constructing various phases of a compiler.

Construction tools

- They are also known as a **compiler- compilers**, **compiler-** generators or translator
- These **tools** use specific language or algorithm for specifying and implementing the component of the **compiler**

1. Parser generators.
2. Scanner generators.
3. Syntax-directed translation engines.
4. Data-flow analysis engines.
5. Automatic code generators.

Parser Generators

Input: Grammatical description of a programming language

Parser generator takes the grammatical Description(BNF) of a programming language produces a syntax analyzer

stmt : **if** '(' *expr* ')' *stmt* [**else** *stmt*]
| **while** '(' *expr* ')' *stmt*
| **for** '(' [*assg*] ';' [*expr*] ';' [*assg*] ')' *stmt*
| **return** [*expr*] ';'
| *assg* ';'
| **id** '(' [*expr* [',' *expr*]] ')' ';'
| '{' { *stmt* } '}'
| ';'

assg : **id** ['[' *expr* ']'] = *expr*

expr : '-' *expr*
| '!' *expr*
| *expr binop expr*
| *expr relop expr*
| *expr logical_op expr*
| **id** ['(' [*expr* [',' *expr*]] ')' | '[' *expr* ']']
| '(' *expr* ')'

Production Rule	Meaning
$S \rightarrow NP\ \ VP$	A sentence is a noun phrase followed by a verb phrase
$NP \rightarrow DT\ \ NN$	A noun phrase is a determiner followed by a noun
$NP \rightarrow NP\ \ PP$	A noun phrase can also be another noun phrase followed by a prepositional phrase
$VP \rightarrow Vi$	A verb phrase is simply an intransitive verb
$VP \rightarrow Vt\ \ NP$	A verb phrase is a transitive verb followed by a noun phrase
$VP \rightarrow VP\ \ PP$	A verb phrase can also be another verb phrase followed by a prepositional phrase
$PP \rightarrow IN\ \ NP$	A prepositional phrase is a preposition followed by a noun phrase
$DT \rightarrow$ the	The is a determiner
$NN \rightarrow$ dog | cat | tree	Dog, cat, and tree are nouns
$IN \rightarrow$ on	On is a preposition
$Vt \rightarrow$ chased | perched	Both chased and perched are transitive verbs
$Vi \rightarrow$ sneeze	Sneeze is an intransitive verb

The generated code is a *parser*, which takes a sequence of characters and tries to match the sequence against the grammar

- Antlr and yacc are popular parser generators

Parser generators for C/C++: Bison (in HTML), Bison , **Yacc**

Scanner Generators

Input: Regular expression description of the tokens of a language

Output: Lexical analyzers.

Scanner generator generates lexical analyzers from a regular expression description of the tokens of a language

- Describing tokens(RE)
- Recognizing (or accepted) -via mathematical model called finite-state acceptor

Scanner generators for C/C++: Flex , **Lex**

Syntax-directed Translation Engines

Grammar + semantic rule = SDT (syntax directed translation)

- Every non-terminal can get one or more than one attribute based on the type of the attribute
- The value of these attributes is evaluated by the semantic rules associated with the production rule
- E -> E+T { E.val = E.val + T.val } PR#1
- E -> T { E.val = T.val } PR#2
- T -> T*F { T.val = T.val * F.val } PR#3
- T -> F { T.val = F.val } PR#4
- F -> INTLIT { F.val = INTLIT.lexval } PR#5

Data-flow Analysis Engines

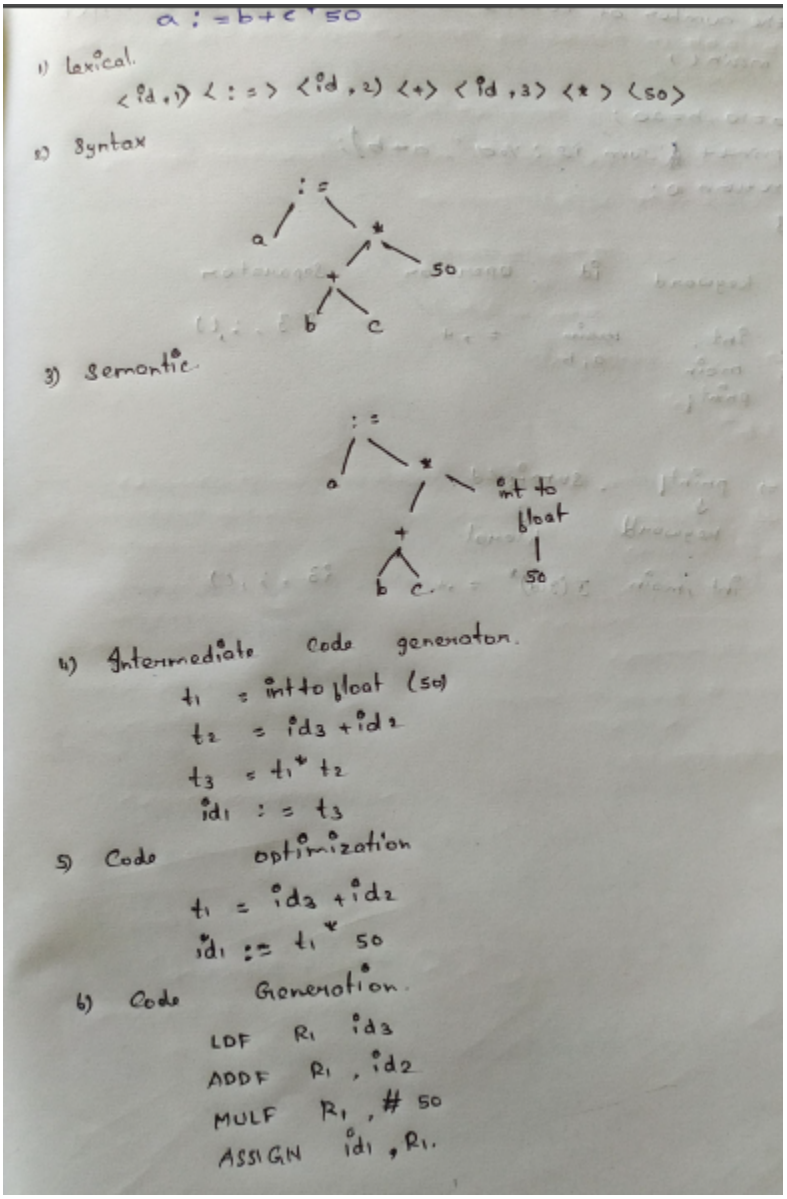
Data-flow analysis is a key part of code optimization.

- These are techniques that derive information about the flow of data along program execution paths
- There is an infinite number of paths through a program and there is no bound on the length of a path
- Data-flow analysis engine gathers the information on values transmitted from one part of a program to each of the other parts
- Highly used for debugging

Automatic Code Generators

Input: Intermediate language.
Output: Machine language.
Code-generator takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for a target machine.

Discuss the working of various phases of Compiler. Interpret the output for each phases for the expression a: = b + c * 50.



A compiler can choose one of the two options.

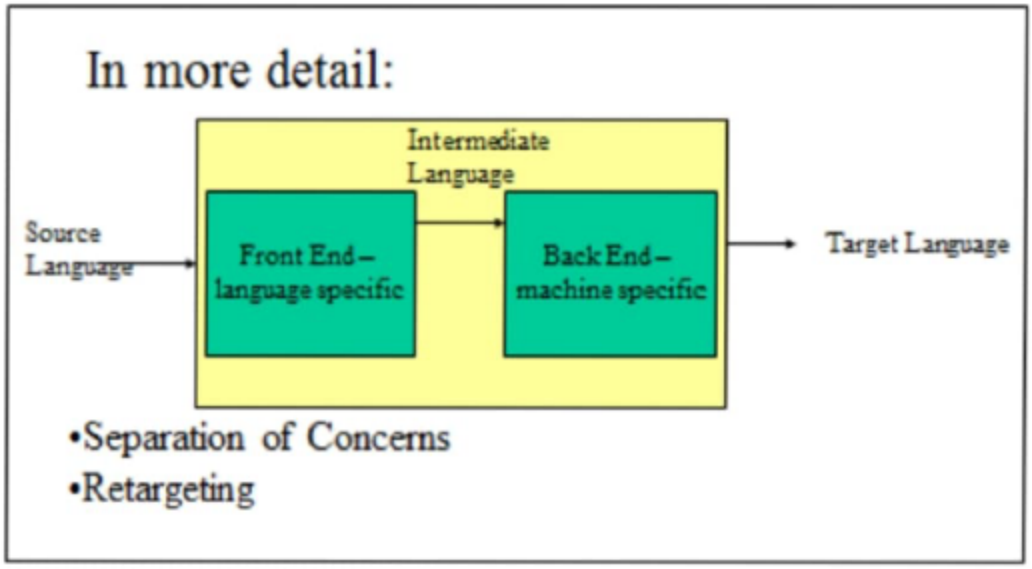
a) Translate the input source into intermediate code and then convert it to final machine code.

b) Directly generate the final machine code from the input source.

What is the preferred option and why? Justify this with the suitable claims.

Preferred option is a

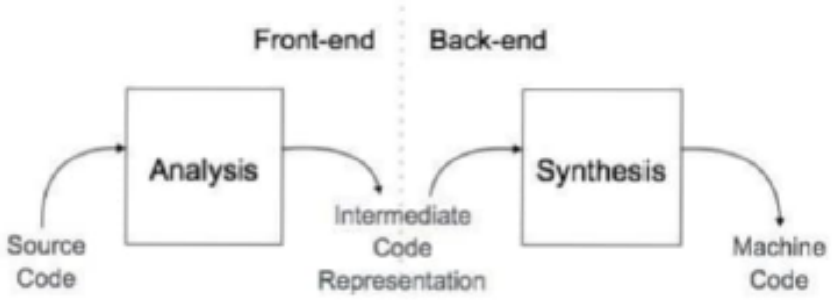
Compiler Architecture



A compiler can broadly be divided into two phases based on the way they compile.

Analysis Phase

Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts, and then checks for lexical, grammar, and syntax errors. The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.



Synthesis Phase

Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

A compiler can have many phases and passes.

- **Pass** : A pass refers to the traversal of a compiler through the entire program.
- **Phase** : A phase of a compiler is a distinguishable stage, which takes input from the previous stage, processes and yields output that can be used as input for the next stage. A pass can have more than one phase.

```
int fact;
```

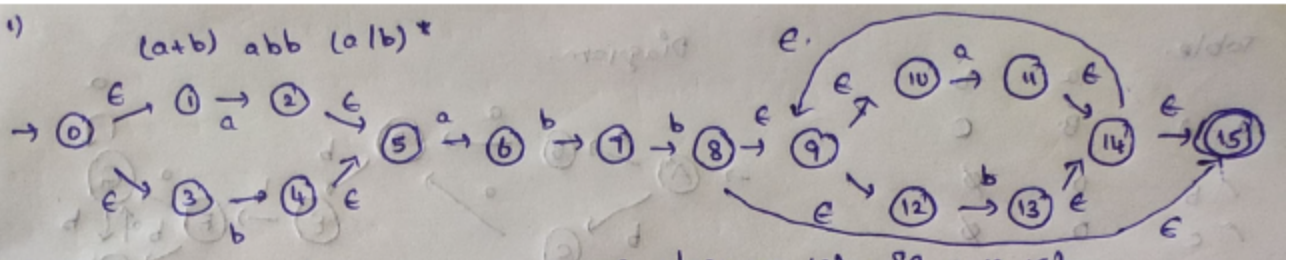
```
int factorial (int n)
{
     int val;
if(n>1){
   val=n*factorial (n-1);
return (val);
}
else
{
return(1);
}
}
int main()
{ printf (“factorial program”);
Fact 5=factorial(5);
Printf(“fact=5=%d \n”, fact5);
}
```

Consider the above program, What are the data structures used to store the information, how the symbol table management is handled?

Describe the need of separating the analysis phase into lexical phase and parsing?

Solve the given regular expression (a+b) abb (a/b)* into NFA using Thompson construction and then to minimized DFA. . Describe the sequence of moves made by each in processing the input string ababba. Explain in detail the structure of the compilers. Compare language dependent and independent phases

1) $(a+b)\ abb\ (a|b)^*$



ε-closure (0) = {0,1,3}
ε-closure (1) = {1}
ε-closure (2) = {2,5}
ε-closure (3) = {3}
ε-closure (4) = {4,5}
ε-closure (5) = {5}
ε-closure (6) = {6}
ε-closure (7) = {7}
ε-closure (8) = {8,9,10,12,15}
ε-closure (0) = {0,1,3} = A

ε-closure (9) = {9,10,12,15}
ε-closure (10) = {10}
ε-closure (11) = {11,14,15,9,10,12}
ε-closure (12) = {12}
ε-closure (13) = {13,14,9,10,12}
ε-closure (14) = {14,9,10,12,15}
ε-closure (15) = {15}.

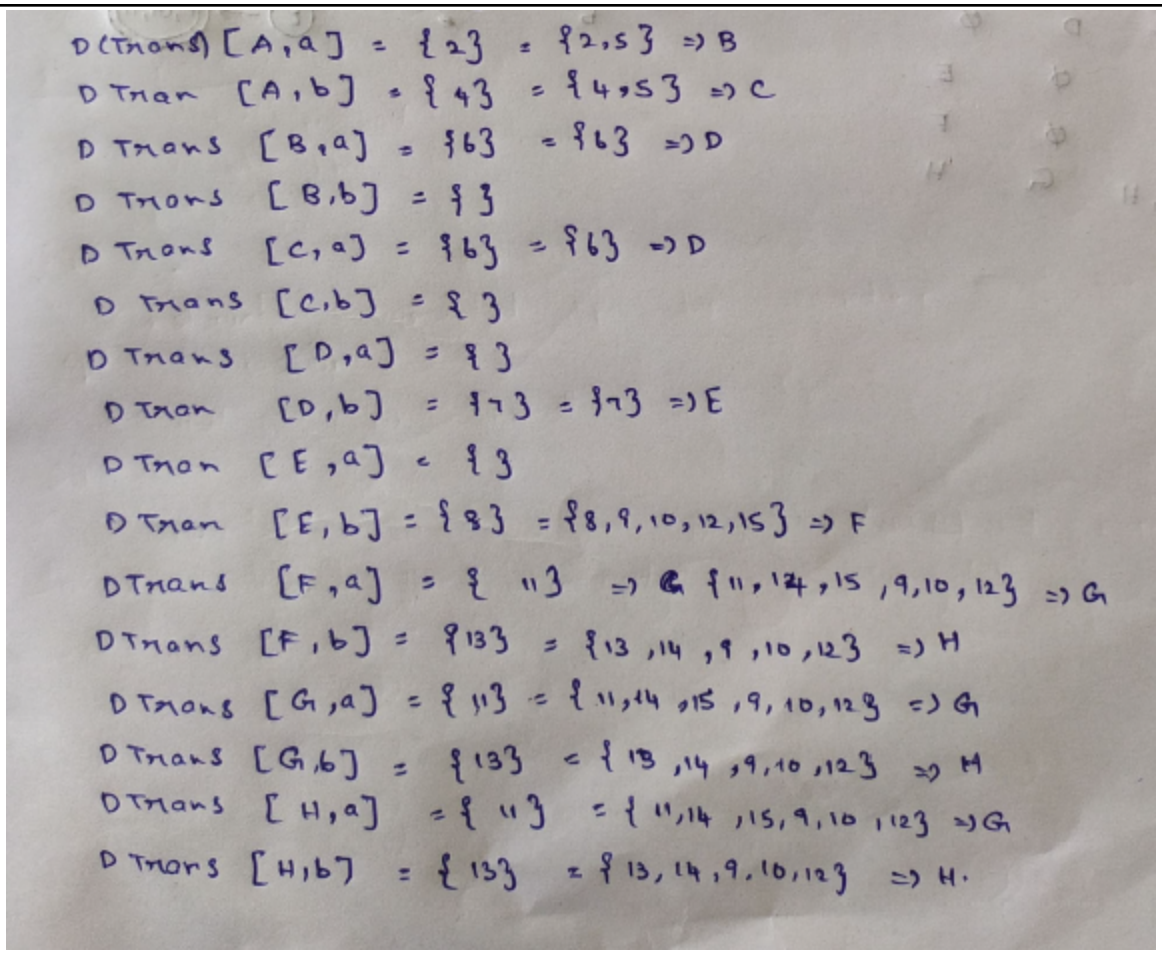

D(Trans) [A,a] = {2} = {2,5} => B
D Tran [A,b] = {4} = {4,5} => C
D Trans [B,a] = {6} = {6} => D
D Trans [B,b] = { }
D Trans [C,a] = {6} = {6} => D
D Trans [C,b] = { }
D Trans [D,a] = { }
D Tran [D,b] = {7} = {7} => E
D Tran [E,a] = { }
D Tran [E,b] = {8} = {8,9,10,12,15} => F
DTrans [F,a] = {11} => {11,14,15,9,10,12} => G
DTrans [F,b] = {13} = {13,14,9,10,12} => H
DTrans [G,a] = {11} = {11,14,15,9,10,12} => G
DTrans [G,b] = {13} = {13,14,9,10,12} => H
DTrans [H,a] = {11} = {11,14,15,9,10,12} => G
D Trans [H,b] = {13} = {13,14,9,10,12} => H.

Table.

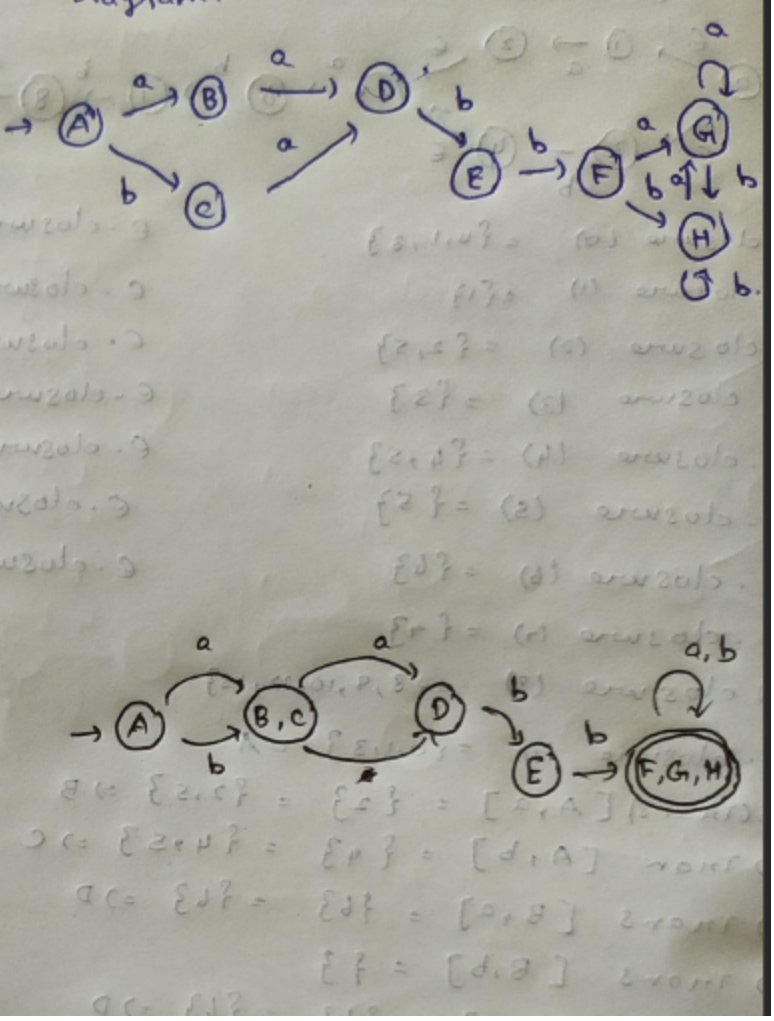
	a	b
→ A	B	C
^ B	D	∅
^ C	D	∅
D	∅	E
E	∅	F
^ F	G	H
^ G	G	H
* ^ H	G	H

↓

Minimization.

	a	b
A	B	C
B,C	D	∅
D	∅	E
E	∅	F
F,G,H	G	H

Diagram.



Compare tokens, patterns and lexemes

Lexeme

A lexeme is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

Token

The token is a sequence of characters which represents a unit of information in the source program.

Pattern

A pattern is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

Count the number of tokens in the given snippet.

```
int main()
{
a=10,b=20;
printf(“ sum is : %d”, a+b;
return 0;
}
```

Explain various Errors encountered in different phases of compiler.

LEXICAL ERRORS :-

→ incorrect or misspelled name of some identifier

SYNTACTICAL ERRORS

→ Missing semicolon, unbalanced parenthesis

→ When an error is detected, it must be handled by parser to enable the parsing of the rest of the input.

→ Most of the errors occured are syntactic errors

→ Role of the Error handler in parser are

*) Report the presence of errors clearly & accurately

*) Recover from each error quickly enough to detect subsequent errors

*) Add minimal overhead to the processing of correct programs.

→ 4 - common Error recovery Strategies implemented in the parser to deal with the error in the code

*) panic mode
*) Statement level
*) Error productions
*) Global correction

SEMANTIC ERRORS

→ Incompatible value assignment

→ semantic Analyzer corrects the foll. errors

*) Type mismatch
*) undeclared variable
*) Reserved identifier misuse
*) Multiple declaration of a variable in a scope
* Accessing an out of scope variable
* Actual & formal Parameter mismatch

Lexical phase errors

These errors are detected during the lexical analysis phase. It occurs when compiler does not recognize valid token string while scanning the code.

- Spelling error
- Exceeding length of identifier or numeric constants.
- Appearance of illegal characters
- Unmatched string
- Transposition of two characters

Example: 1- **This is a comment */**
Example 2:

```
void main()
{
int x=10, y=20;
char * a;
a= &x;
x= 1xab;
}
```

Error recovery:

Panic Mode Recovery

successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are delimiters such as; or

Syntactic phase errors

These errors are detected during syntax analysis phase. Typical syn

- Errors in structure
- Missing operator
- Misspelled keywords
- Unbalanced parenthesis

```
Example : swicth(ch)
          {
             .......
             .......
          }
```

Lexical error: *I took a bite of an appel*

Correction: *I took a bite of an apple.*

Hence, **"Unidentified keyword/identifier"** error occurs.

Semantic errors

-occurs when a statement is syntactically valid, but does not do what the **programmer** intended without producing **error** messages

Example :

- An expression may not be evaluated in the order you expect, yielding an incorrect result
- Undeclared variables(symbol table entry has to be made for error recovery)

```
Example : int a[10], b;
                .......
                .......
                a = b;
```

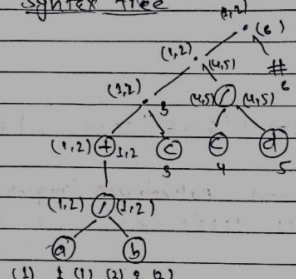
Example: Jack left her homework at home

A “type mismatch” between her and Jack; we know they are different people

Obtain the required DFA from the regular expression (a/b)* a (a/b) using Minimization of DFA using Π new constructions. Write down the suitable algorithm
Explain in detail the structure of the compilers. Compare language dependent and independent phases.
Construct transition diagram for keyword, identifiers and relational operators

Is it possible to directly obtain a DFA from a regular expression? How is it different from obtaining DFA from an intermediate NFA? Obtain DFA directly for the regular expression $(a/b)^{+}c(c/d)$.

Syntax tree



Node	Follow (i)
1	{1, 2, 3}
2	{1, 2, 3}
3	{4, 5}
4	{6}
5	{6}
6	ϕ

Now, we construct DFA for the given regular expression. The start states of DFA is firstpos of root.

firstpos of root = {1, 2, 3}

let the start be A.

A = {1, 2}

Consider the symbol 'a' on set A.

position 1 has a transition.

followpos(1)

⇒ {1, 2, 3}

⇒ B

Consider the input symbol 'b' on set A.

position 2 has a transition.

followpos(2)

⇒ {1, 2, 3}

⇒ B

Now, set A has no position left for c & d. so, on input c & d, it will go to dead state (Z_d).

Scanned with CamScanner

D.Trans [A, a] = B
D.Trans [A, b] = B
D.Trans [A, c] = Z_d
D.Trans [A, d] = Z_d

Again,

Consider the symbol 'a' on set B.

position 1 has a transition.

followpos(1)

⇒ B.

Consider the symbol 'b' on set B.

followpos(2)

⇒ B

Consider the symbol 'c' on set B.

followpos(3)

⇒ {4, 5}

⇒ C

∴

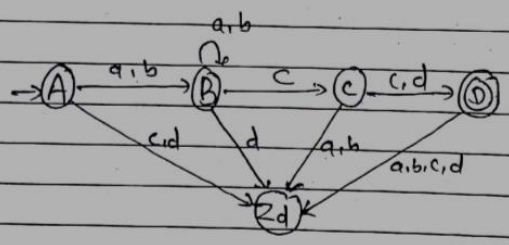
D.Trans [B, a] = B
D.Trans [B, b] = B
D.Trans [B, c] = C
D.Trans [B, d] = Z_d

Again,
Consider the symbol 'c' on set C.
followpos(c)
=> {.6 }
=> D
Consider the symbol d on set C
followpos (.5)
=> {6} = D

D.Tran [C, a] = Zd
D.Tran [C, b] = Zd
D.Tran [C, c] = D
D.Tran [C, d] = D

Note:
∴ On input any symbol on set D, it will go to dead state (Zd).

DfA Diagram:



Dead State
or
Trap state

Hence, this is the required DfA for given regular expression.

How do we eliminate ambiguity from the dangling else grammar?

Date : / /

Eliminating Ambiguity :

An ambiguous Gr can be rewritten to eliminate the ambiguity.

Ambiguous Gr cannot distinctively determine which parse tree to select for a sentence or string

Hence disambiguiting rules are used with ambiguous Gr to do away with undesirable pause trees thereby leaving only one tree for a sentence

ex) removing ambiguity from the "dangling else" Gr

Stmt → if expr then stmt
| if expr then stmt else stmt
| other

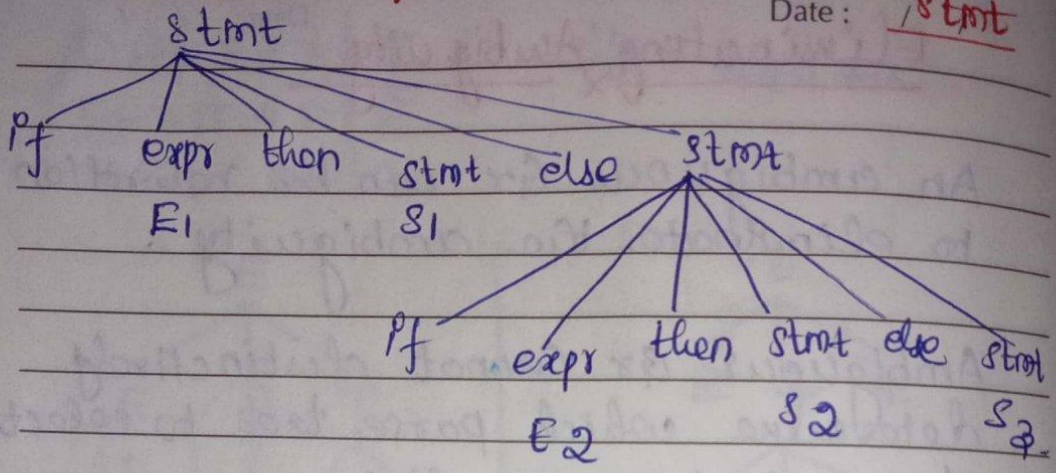
Other → any other statement.

Consider the compound conditional stmt,

If E_1 then S_1 else if E_2 then S_2 else S_3

Parse Tree for the Above conditional

Date :



ex : 2

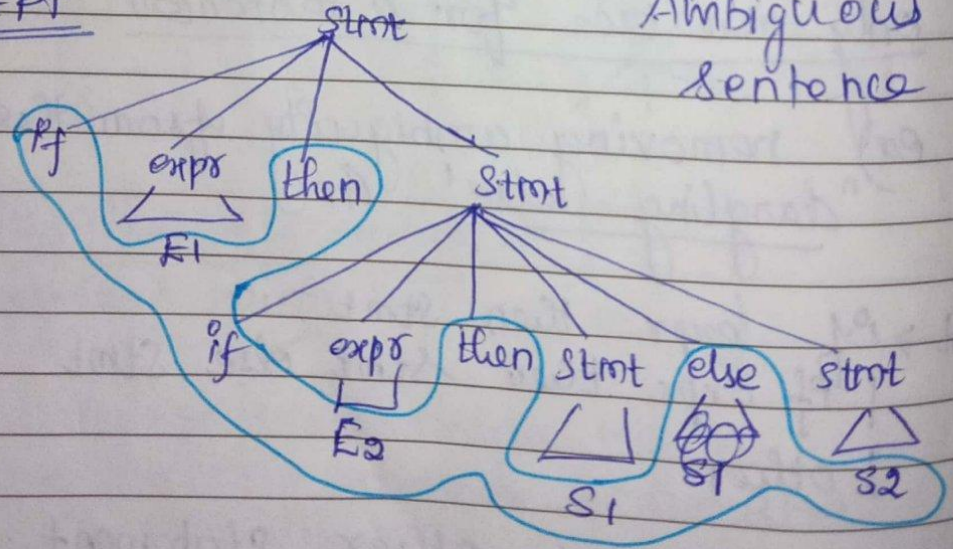
pf E1 then pf E2 then S1 else S2

It has two PT's

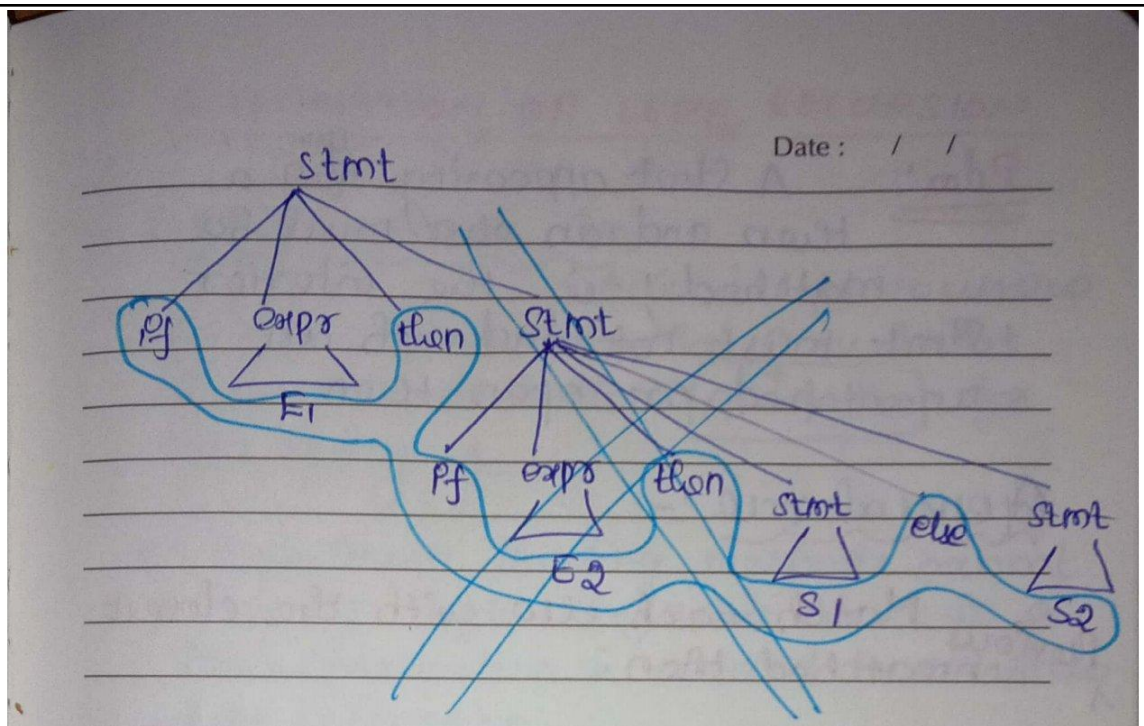
⇓

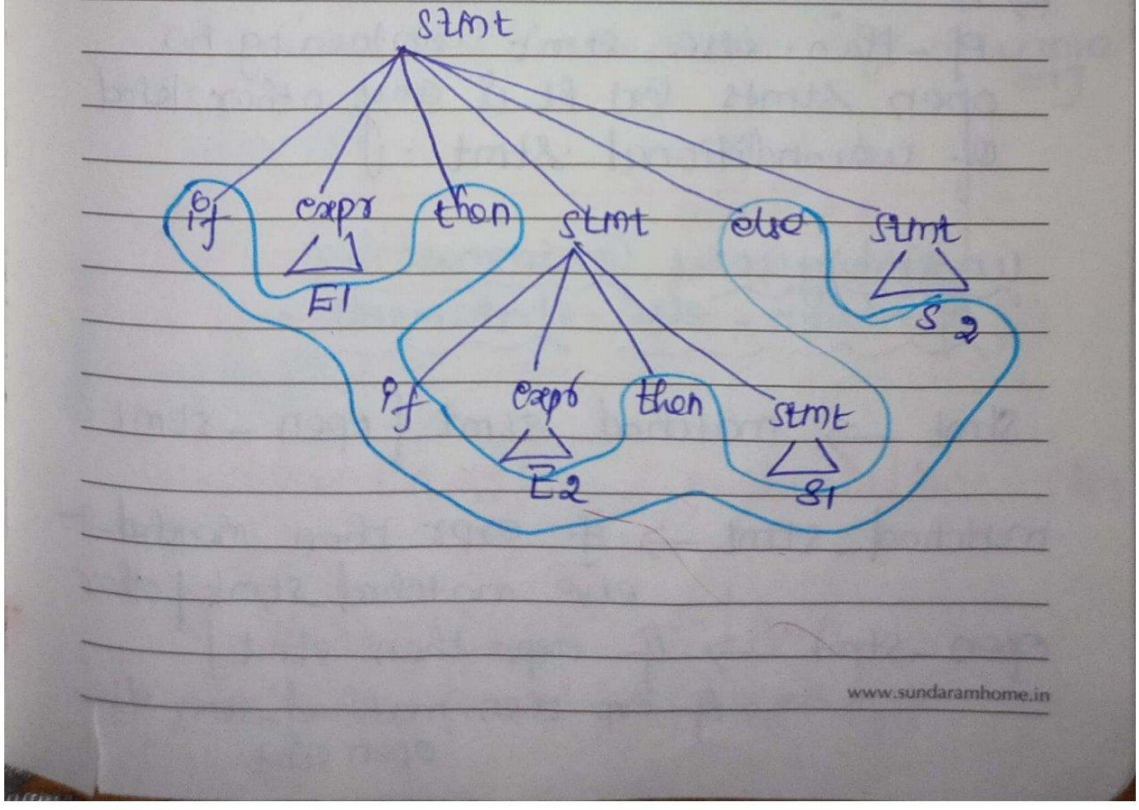
Ambiguous Sentence

1st PT



Date : / /





Date : / /

Idea: A Stmt appearing b/w a then and an else must be matched; ie the interior Stmt must not end with an unmatched or open then.

General rule :-

Match each else with the closest previous unmatched then.

↳ A matched Stmt is either an if-then-else Stmt containing no open stmts (or) it is any other kind of unconditional Stmt.

Unambiguous Grammar for if-then-else statements :-

Stmt → matched_stmt | open_stmt

matched_stmt → if expr then matched_stmt else matched_stmt | other

open_stmt → if expr then stmt | if exp then matched_stmt else open stmt

Explain the structure of Lex program. Write Lex specifications for a Desk calculator application.

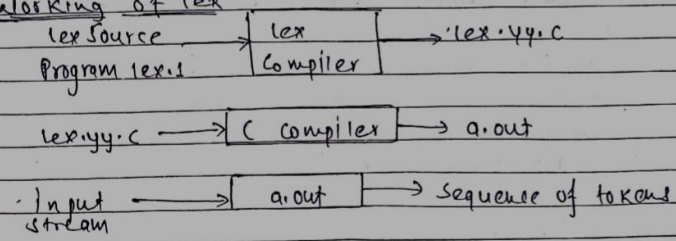
⇒ Lex is a program that generates lexical analyzer. It is used with YACC parser generator. The lexical analyzer is a program that transforms an input stream into a sequence of tokens. It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

Structure of lex program:

A lex program is seperated into three sections by %% delimiters.

```
{ definitions }
%%
{ rules }
%%
{ user subroutines }
```

Working of lex



lex program to implement a calculator.

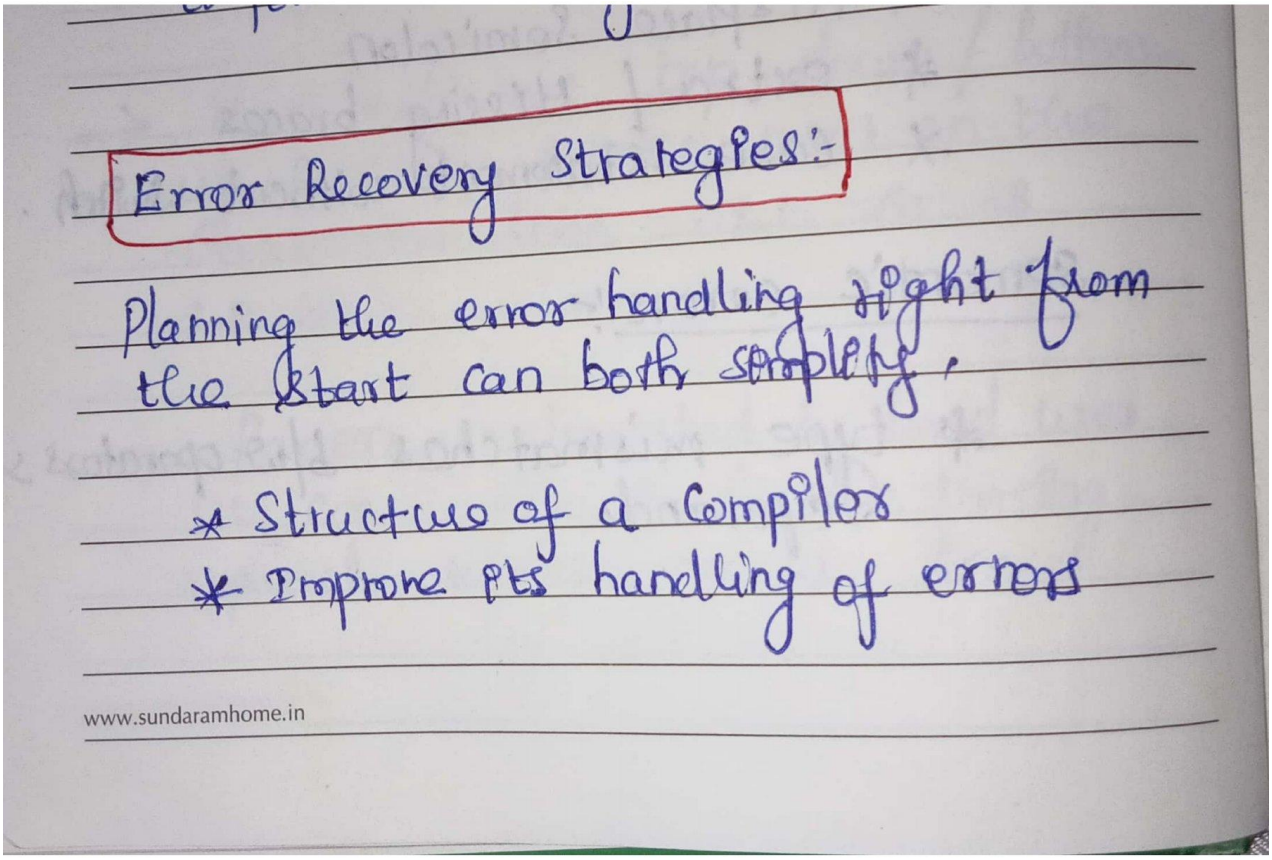
```
%{
  int op = 0,i;
  float a,b;
%}
dig [0-9]+([0-9]*)"."([0-9]+)
add "+"
sub "-"
mul "*"
div "/"
pow "^"
in \n
%%
/* digi() is a user defined function */
{dig} {digi();}
{add} {op=1;}
{sub} {op=2;}
{mul} {op=3;}
{div} {op=4;}
{pow} {op=5;}
```

```
/* digi() is a user defined function */
{dig} {digi(); }
{add} {op=1; }
{sub} {op=2; }
{mul} {op=3; }
{div} {op=4; }
{pow} {op=5; }
{ln} {printf("\n The Answer : %f\n\n", a); }
%%
digi()
{
if(op=0)
  a = atof(yytext);
switch(op)
 {
```



```
case 1: a=a+b;
    break;
case 2: a=a-b;
    break;
case 3: a=a*b;
    break;
case 4: a=a/b;
    break;
case 5: for(i=a; b>1; b--)
    a=a*i;
    break; }
}
  op=0;
 }
}
main(int argv, char *argc[])
 {
    yylex(); }
  yywrap()
  {
    return 1; }
  }
```

Describe the different strategies that a parser can employ to recover from a syntactic error.



Error Recovery Strategies:-

Planning the error handling right from the start can both simplify.

* Structure of a Compiler
* Improve its handling of errors

www.sundaramhome.in

Date : / /

Error handler in a parser has the following goals :-

* Report the presence of errors clearly & accurately
* Recover from each error quickly enough to detect subsequent errors.
* Add the minimal overhead to the processing of correct programs.

Four Error Handling strategies :-

* Panic mode
* phrase ~~Mode~~ Level Recovery
* Error production
* Global correction.

① Panic Mode :-

* easiest method, prevents the parser from infinite loops

Date : / /

* when parser finds an error in the start, it ignores the rest of the stmt by not processing the input.

* when an err is found, all i/ps are discarded until it finds a synchronizing token.
  ↳ delimiters, semicolon

Adv	DisAdv
* simplicity * no loops formed	Additional errors are not checked since i/ps are skipped

② phrase x level Recovery

On discovering an error, Parser performs local correction on the remaining i/p.

Eg)

* Replacing a Prefix by some str
* Replacing comma by semicolon
* Deleting extraneous semicolon
* Insert Missing semicolon.

Date : / /

* by performing local corrections, allow the parser to continue.

* one wrong correction will lead to an infinite loop.

Adv	Disadv
It can correct any i/p string	correction should not lead to infinite loop.
	If the error has occured b/f the point of detection

⑤ Error Production

⇒ Productions which generate erroneous constructs are augmented to the Gr by considering common errors that occur

⇒ These productions detect the anticipated errors during parsing.

⇒ Error diagnostics about the erroneous constructs are generated by the parser

Consider the following CFG G = (N={S, A, B, C, D}, T={a,b,c,d}, P, S) where the set of productions P is given below:

$S \rightarrow A$

$A \rightarrow BC \mid DBC$

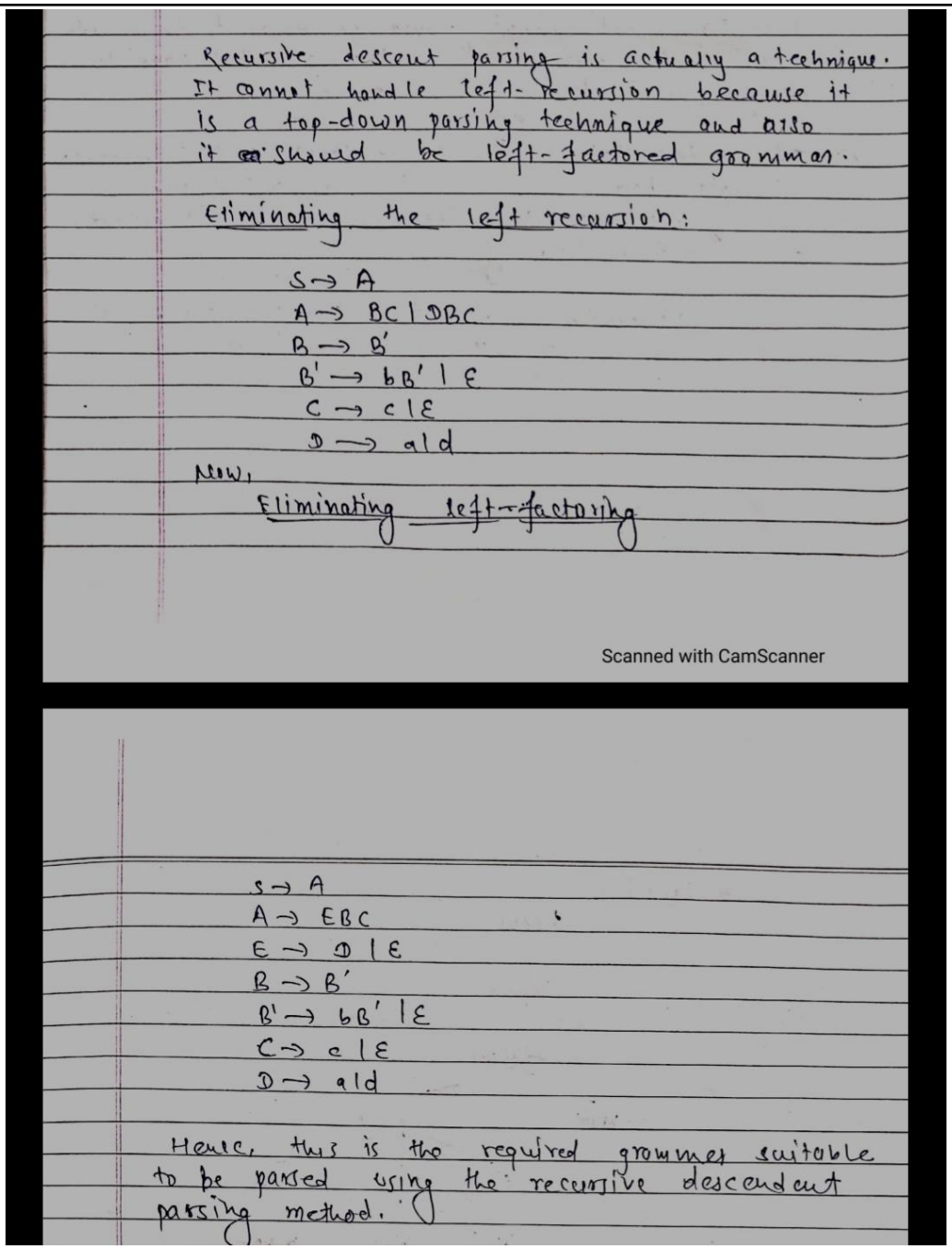
$B \rightarrow Bb \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow a \mid d$

Is this grammar suitable to be parsed using the recursive descendent parsing method?

Justify and modify the grammar if needed



Recursive descent parsing is actually a technique. It cannot handle left-recursion because it is a top-down parsing technique and also it should be left-factored grammar.

Eliminating the left recursion:

S → A
A → BC | DBC
B → B'
B' → bB' | ε
C → c | ε
D → a | d

Now,

Eliminating left-factoring

Scanned with CamScanner

S → A
A → EBC
E → D | ε
B → B'
B' → bB' | ε
C → c | ε
D → a | d

Hence, this is the required grammer suitable to be parsed using the recursive descendent parsing method.

Consider the grammar

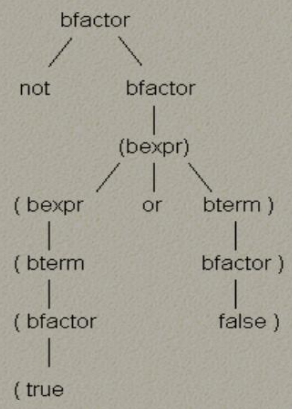
bexpr -> bexpr or bterm | bterm
bterm -> bterm and bfactor | bfactor
bfactor -> not bfactor | (bexpr) | true | false

a) Construct a parse tree for the input string **not (true or false)**Show that this grammar generates all boolean expressions.

bexpr ::= bexpr **or** bterm | bterm
bterm ::= bterm **and** bfactor | bfactor
bfactor ::= **not** bfactor | (bexpr) | **true** | **false**

Construct a parse tree for the sentence **not** (**true or false**).
Show that this grammar generates all Boolean expressions

Parse Tree for: **not (true or false)**

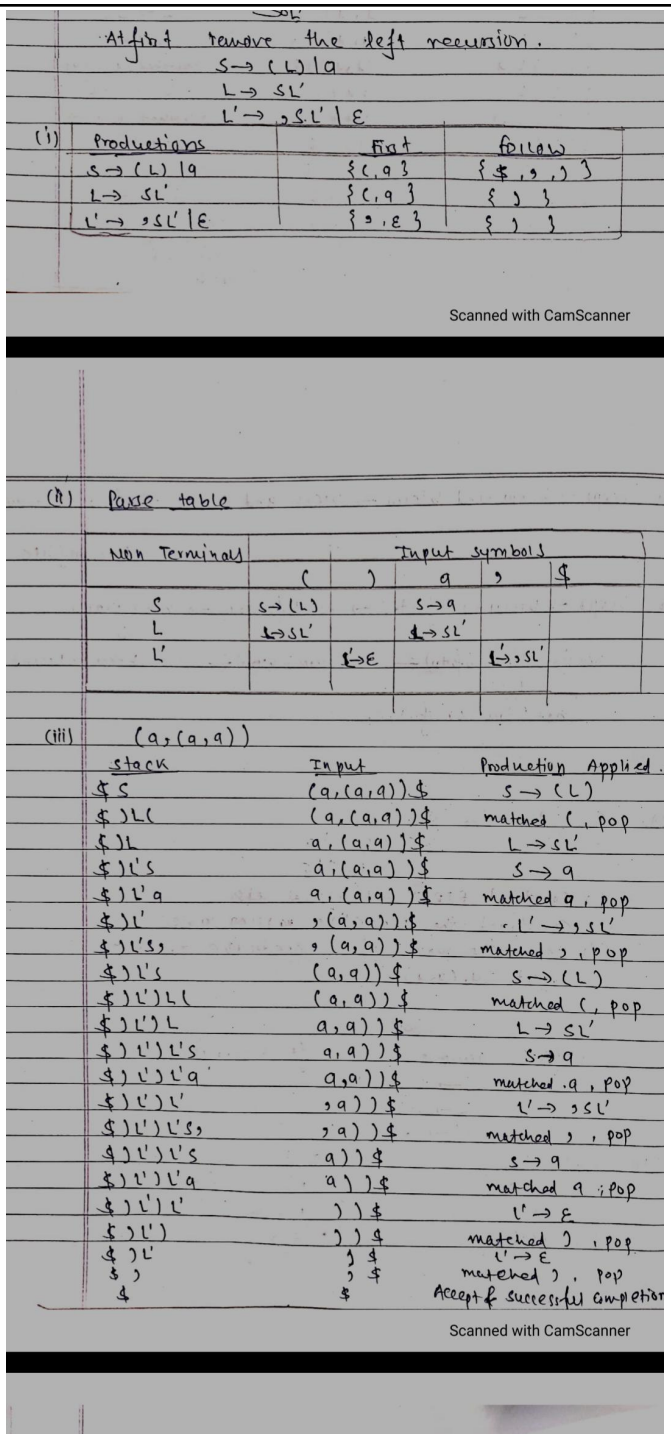


Proof that this language supports all Boolean expressions:

bfactor=> bfactor => TRUE	bfactor=> bfactor => FALSE
bfactor r=>bfactor => not bfactor => not TRUE	bfactor=>bfactor => not bfactor => not FALSE
nor:	**nand:**
bfactor => not bfactor => not (bexpr) => not (bexpr or bterm) => not (bterm or bterm) => not (bfactor or bterm) => not (true or bterm) => not (true or bfactor) => not (true or false)	bfactor => not bfactor => not (bexpr) => not (bterm) => not (bterm and bfactor) => not (bfactor and bfactor) => not (true and bfactor) => not (true and false)
xor:	**xnor:**
bexpr => bexpr or bterm => bterm or bterm => bfactor or bterm => (bexpr) or bterm => (bterm) or bterm => (bterm and bfactor) or bterm => (bfactor and bfactor) or bterm => (true and bfactor) or bterm => (true and false) or bterm => (true and false) or bfactor => (true and false) or (bexpr) => (true and false) or (bexpr or bterm) => (true and false) or (bterm or bterm) => (true and false) or (bfactor or bfactor) => (true and false) or (false and bfactor) => (true and false) or (false and true)	bexpr => bexpr or bterm => bterm or bterm => bfactor or bterm => (bexpr) or bterm => (bterm and bfactor) or bterm => (bfactor and bfactor) or bterm => (true and bfactor) or bterm => (true and false) or bterm => (true and false) or bfactor => (true and false) or (bexpr) => (true and false) or (bexpr or bterm) => (true and false) or (bterm or bterm) => (true and false) or (bfactor or bterm) => (true and false) or (false or bterm) => (true and false) or (false or bfactor) => (true and false) or (false or true)

NOT is implemented by bfactor
OR is implemented by bterm
AND is implemented by bexpr

•
Illustrate the heuristic techniques for error-recovery in predictive parsing
Consider the grammar $S \rightarrow (L) \mid a$ $L \rightarrow L, S \mid S$ i) Construct FIRST and FOLLOW sets ii)Construct the predictive parsing table iii)Show the moves of the predictive parser for input (a,(a,a))



At first remove the left recursion.

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \varepsilon$

(i)

Productions	First	Follow
$S \rightarrow (L) \mid a$	$\{(, a\}$	$\{\$, ,,)\}$
$L \rightarrow SL'$	$\{(, a\}$	$\{)\}$
$L' \rightarrow ,SL' \mid \varepsilon$	$\{,, \varepsilon\}$	$\{)\}$

Scanned with CamScanner

(ii) Parse table

Non Terminals	Input symbols				
	(	)	a	,	$
S	$S \rightarrow (L)$		$S \rightarrow a$		
L	$L \rightarrow SL'$		$L \rightarrow SL'$		
L'		$L' \rightarrow \varepsilon$		$L' \rightarrow ,SL'$	

(iii) (a,(a,a))

Stack	Input	Production Applied
$S	(a,(a,a))$	$S \rightarrow (L)$
$)L(	(a,(a,a))$	matched (, pop
$)L	a,(a,a))$	$L \rightarrow SL'$
$)L'S	a,(a,a))$	$S \rightarrow a$
$)L'a	a,(a,a))$	matched a, pop
$)L'	,(a,a))$	$L' \rightarrow ,SL'$
$)L'S,	,(a,a))$	matched , pop
$)L'S	(a,a))$	$S \rightarrow (L)$
$)L')L(	(a,a))$	matched (, pop
$)L')L	a,a))$	$L \rightarrow SL'$
$)L')L'S	a,a))$	$S \rightarrow a$
$)L')L'a	a,a))$	matched a, pop
$)L')L'	,a))$	$L' \rightarrow ,SL'$
$)L')L'S,	,a))$	matched , pop
$)L')L'S	a))$	$S \rightarrow a$
$)L')L'a	a))$	matched a, pop
$)L')L'	))$	$L' \rightarrow \varepsilon$
$)L')	))$	matched), pop
$)L'	))$	$L' \rightarrow \varepsilon$
$)	))$	matched), pop
$	$	Accept & successful completion

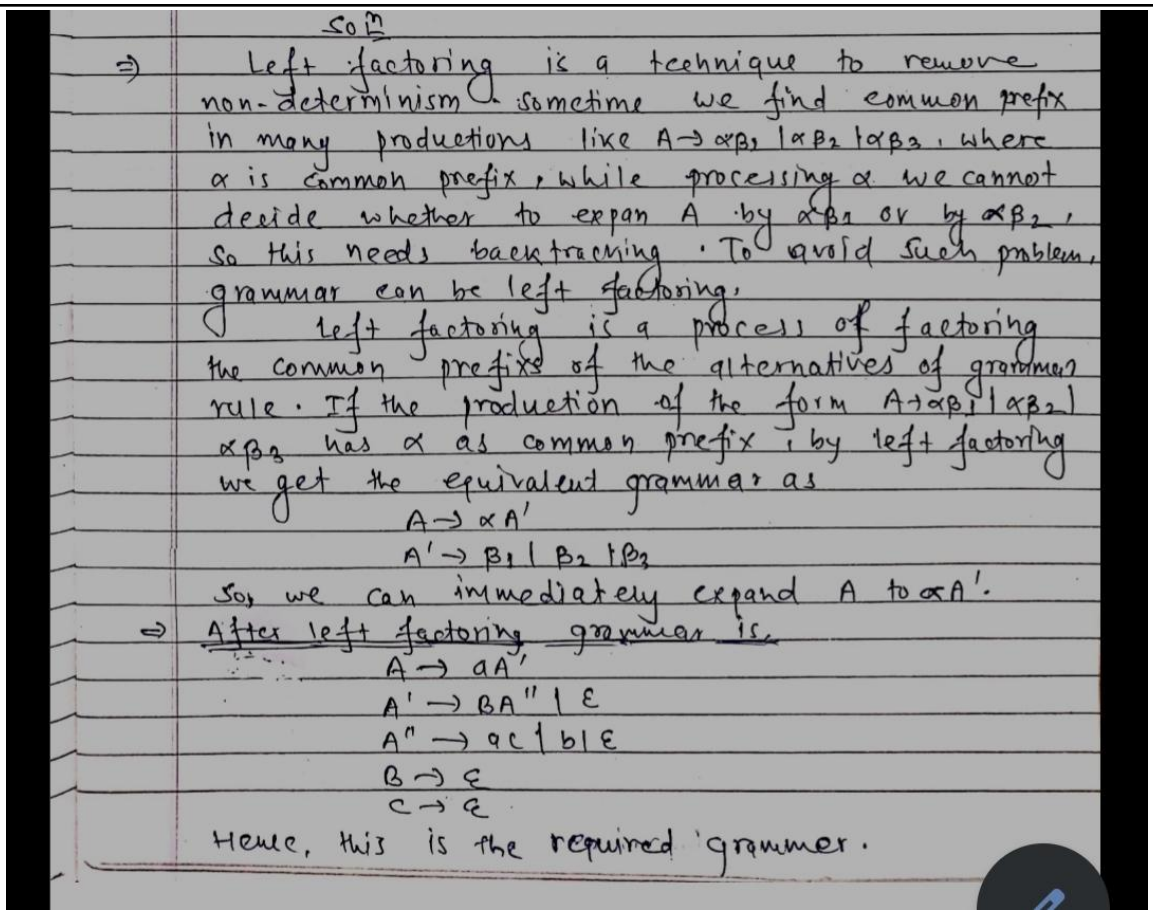
Scanned with CamScanner

Left factoring is a technique to remove non-determinism. How left factoring does avoid backtracking in grammars? Apply left factoring in the following grammar to make it deterministic.

$A \rightarrow aBcC \mid aBb \mid aB \mid a$

$B \rightarrow \varepsilon$

$C \rightarrow \varepsilon$



Soln

⇒ Left factoring is a technique to remove non-determinism. Sometime we find common prefix in many productions like $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$. where α is common prefix, while processing α we cannot decide whether to expan A by $\alpha\beta_1$ or by $\alpha\beta_2$, so this needs backtracking. To avoid such problem, grammar can be left factoring.

Left factoring is a process of factoring the common prefixes of the alternatives of grammar rule. If the production of the form $A \to \alpha\beta_1 | \alpha\beta_2 | \alpha\beta_3$ has α as common prefix, by left factoring we get the equivalent grammar as

$A \to \alpha A'$

$A' \to \beta_1 | \beta_2 | \beta_3$

So, we can immediately expand A to $\alpha A'$.

⇒ After left factoring grammar is,

$A \to aA'$

$A' \to BA'' | \varepsilon$

$A'' \to ac | b | \varepsilon$

$B \to \varepsilon$

$C \to \varepsilon$

Hence, this is the required grammer.

How do SLR(1), LR(1) and LALR(1) methods compare against each other in the process of constructing the parsing table from the grammar?

SLR Parser

SLR represents "Simple LR Parser". It is very easy and cost-effective to execute. The SLR parsing action and goto function from the deterministic finite automata that recognizes viable prefixes. It will not make specificially defined parsing action tables for all grammars but does succeed on several grammars for programming languages. Given a grammar G. It augment G to make G', and from G' it can construct C, the canonical collection of a set of items for G'. It can construct ACTION the parsing action function, and GOTO, the goto function, from C using the following simple LR Parsing table construction technique. It needed us to understand FOLLOW (A) for each non-terminal A of a grammar.

CLR Parser

CLR refers to canonical lookahead. CLR parsing uses the canonical collection of LR (1) items to construct the CLR (1) parsing table. CLR (1) parsing table make more number of states as compared to the SLR (1) parsing. In the CLR (1), it can locate the reduce node only in the lookahead symbols.

LALR Parser

LALR Parser is Look Ahead LR Parser. It is intermediate in power between SLR and CLR parser. It is the compaction of CLR Parser, and hence tables obtained in this will be smaller than CLR Parsing Table.

For constructing the LALR (1) parsing table, the canonical collection of LR (1) items is used. In the LALR (1) parsing, the LR (1) items with the equal productions but have several look ahead are grouped to form an

individual set of items. It is frequently the similar as CLR (1) parsing except for the one difference that is the parsing table.

The overall structure of all these LR Parsers is the same. There are some common factors such as size, class of context-free grammar, which they support, and cost in terms of time and space in which they differ.

Let us see the comparison between SLR, CLR, and LALR Parser.

SLR Parser	LALR Parser	CLR Parser
It is very easy and cheap to implement.	It is also easy and cheap to implement.	It is expensive and difficult to implement.
SLR Parser is the smallest in size.	LALR and SLR have the same size. As they have less number of states.	CLR Parser is the largest. As the number of states is very large.
Error detection is not immediate in SLR.	Error detection is not immediate in LALR.	Error detection can be done immediately in CLR Parser.

SLR Parser	LALR Parser	CLR Parser
SLR fails to produce a parsing table for a certain class of grammars.	It is intermediate in power between SLR and CLR i.e., SLR ≤ LALR ≤ CLR.	It is very powerful and works on a large class of grammar.
It requires less time and space complexity.	It requires more time and space complexity.	It also requires more time and space complexity.

Consider the grammar

$S \rightarrow S(S)S \mid \varepsilon$

Construct the predictive parsing table for the input (() ())

Consider the following (subset of a) CFG grammar

stmt NIL | stmt ';' stmt | ifstmt | whilestmt | stmt

ifstmt IF bexpr THEN stmt ELSE stmt

IF bexpr THEN stmt

whilestmt WHILE bexpr DO stmt

where NIL, ';', IF, THEN, WHILE and DO are terminals, and "stmt", "ifstmt", "whilestmt" and "bexpr" are non-terminals.

For this grammar answer the following questions:

a) Is it ambiguous? Why? Is that a problem?

b) Rewrite the grammar so as to eliminate ambiguity

Draw the precedence function graph for the following table.

	A	(	)	,	$
a			>	>	>
(	<	<	=	<	
)			>	>	>
,	<	<	>	>	
$	<	<			

For the following three address code, give the quadruple representation

t1 := -r

t2 := q*t1

t3 := -r

t4 := s * t3

t5 := t2 + t4

p := t5

Construct LALR(1) parsers for the following grammar and parse the sentence id=id

S → L = R
S → R
L → * R
L → id
R → L

The LR(0) finite-state machine for G has the following states.

State 0:
S′ → · S
S → · L = R
S → · R
L → · * R
L → · id
R → · L

State 1:
S′ → S ·

State 2:
S → L · = R
R → L ·

State 3:
S → R ·

State 4:
L → * · R
R → · L
L → · * R
L → · id

State 5:
L → id ·

State 6:
S → L = · R
R → · L
L → · * R
L → · id

State 7:
L → * R ·

State 8:
R → L ·

State 9:
S → L = R ·
The FIRST and FOLLOW sets are as follows.

FIRST(S) = {*, id}
FIRST(L) = {*, id}
FIRST(R) = {*, id}

FOLLOW(S) = {$}
FOLLOW(L) = {$, =}
FOLLOW(R) = {$, =}
Look at state 2. LR(0) item S → L · = R calls for a shift on lookahead =. But LR(0) item R → L · calls for a reduce on lookahead =, since FOLLOW(L) contains =. That is a parsing conflict.

But, in reality, = can only follow L when L occurs in the context of production S → L = R, never when L is derived using production R → L.

So the shift reduce conflict in state 2 can be resolved by choosing the shift action without affecting any parses.

Only one occurrence of each object is allowable at a given moment during program execution. Justify your answer with respect to static allocation

Static Storage and Dynamic Allocation

Dynamic memory is controlled by the **new** and **delete** operators, **not** by **scope** and **linkage** rules. So, dynamic memory can be allocated from one function and freed from another function.

Although the storage schemes don't apply to dynamic memory, then **do** apply to **automatic** and **static** pointer variables used to keep track of dynamic memory.

Let's look at the following line of code:

```
int *ptr = new int[10];
```

The 40 bytes of memory allocated by **new** remains in memory until the **delete** frees it. But the pointer **ptr** passes from existence when the function containing this declaration terminates.

If we want to have the 40 bytes of allocated memory available from another function, we need to pass or return its address to that function.

On the other hand, if we declare **ptr** with external linkage, the **ptr** pointer will be available by using:

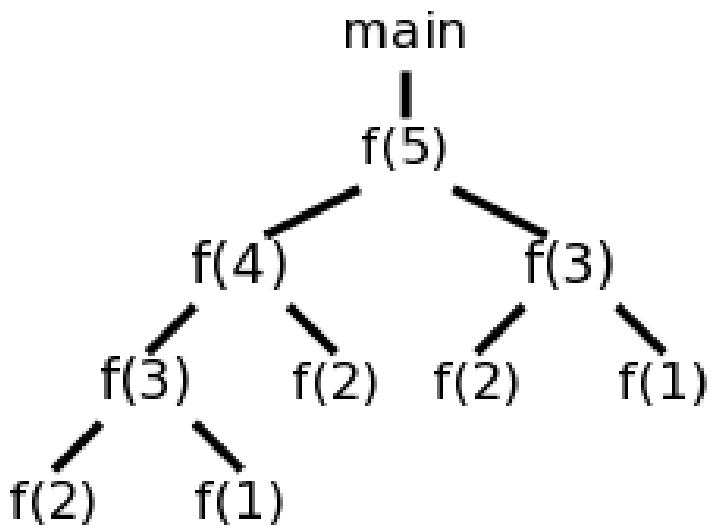
```
extern int *ptr;
```

However, a statement that uses **new** to set **ptr** has to be in a function because static storage variables can only be initialized with constant expressions as shown in the following example.

```
int *ptr;

// initialization with non-const not allowed here

// int *ptr = new int[10];

int main()

{

     ptr = new int[10];

...
```

Design an activation tree for the Fibonacci sequence 1,1,2,3,5,8, ... defined by f(1)=f(2)=1 and, for n>2, f(n)=f(n-1)+f(n-2). Consider the function calls that result from a main program calling f(5).

system starts main
enter f(5)
enter f(4)
enter f(3)
enter f(2)
exit f(2)
enter f(1)
exit f(1)
exit f(3)
enter f(2)
exit f(2)
exit f(4)
enter f(3)
enter f(2)
exit f(2) s
enter f(1)
exit f(1)
exit f(3)
exit f(5)
main ends

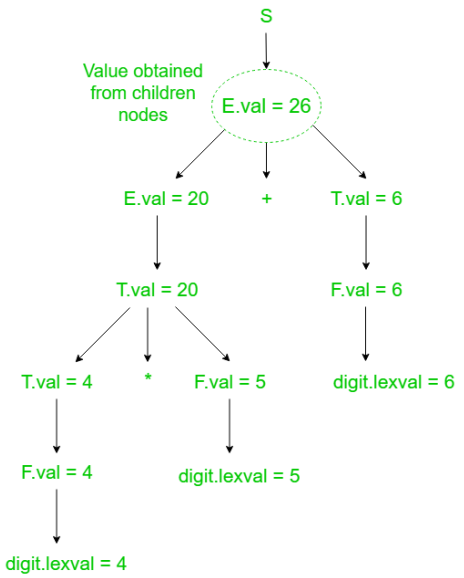


Consider the following grammar
S --> E
E --> E1 + T
E --> T
T --> T1 * F
T --> F
F --> digit
Draw the annotated parse tree to compute s-attribute for the input string 4 * 5 + 6.

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval

Let us assume an input string **4 * 5 + 6** for computing synthesized attributes. The annotated parse tree for the input string is



Annotated Parse Tree

For computation of attributes we start from leftmost bottom node. The rule F –> digit is used to reduce digit to F and the value of digit is obtained from lexical analyzer which becomes value of F i.e. from semantic action F.val = digit.lexval. Hence, F.val = 4 and since T is parent node of F so, we get T.val = 4 from semantic action T.val = F.val. Then, for T –> T1 * F production, the corresponding semantic action is T.val = T1.val * F.val . Hence, T.val = 4 * 5 = 20

Similarly, combination of E1.val + T.val becomes E.val i.e. E.val = E1.val + T.val = 26. Then, the production S –> E is applied to reduce E.val = 26 and semantic action associated with it prints the result E.val . Hence, the output will be 26.

2. Inherited Attributes – These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

Example:

```
A --> BCD   { C.in = A.in, C.type = B.type }
```

Computation of Inherited Attributes –

- Construct the SDD using semantic actions.
- The annotated parse tree is generated and attribute values are computed in top down manner.

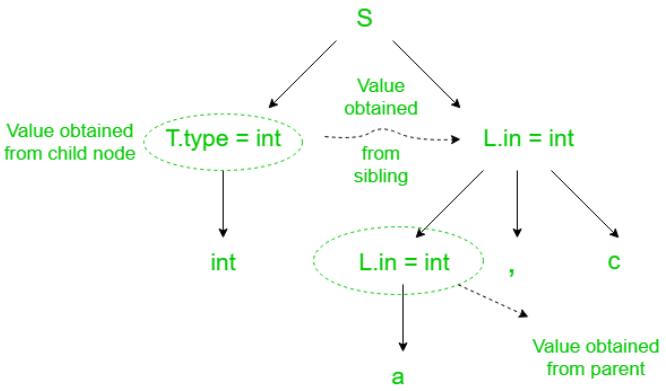
Example: Consider the following grammar

```
S --> T L
T --> int
T --> float
T --> double
L --> L1, id
L --> id
```

The SDD for the above grammar can be written as follow

Production	Semantic Actions
S --> T L	L.in = T.type
T --> int	T.type = int
T --> float	T.type = float
T --> double	T.type = double
L --> L_1 , id	L_1.in = L.in Enter_type(id.entry , L.in)
L --> id	Entry_type(id.entry , L.in)

Let us assume an input string **int a, c** for computing inherited attributes. The annotated parse tree for the input string is



Annotated Parse Tree

The value of L nodes is obtained from T.type (sibling) which is basically lexical value obtained as int, float or double. Then L node gives type of identifiers a and c. The computation of type is done in top down manner or preorder traversal. Using function Enter_type the type of identifiers a and c is inserted in symbol table at corresponding id.entry.

Write the program for dot product of two vectors. Optimize this code by Eliminating the common sub expressions.

```
def dotProduct(vect_A, vect_B):

    product = 0

    # Loop for calculate dot product
    for i in range(0, n):
        product = product + vect_A[i] * vect_B[i]

    return product
if   name   =='  main  ':
    vect_A = [3, -5, 4]
    vect B = [2, 6, 5]
    cross_P = []

# dotProduct function call
    print("Dot product:", end =" ")
    print(dotProduct(vect_A, vect_B))
```

For the following three address code, give the triple and indirect triple implementation

t1 := -r

t2 := q*t1

t3 := -r

t4 := s * t3

t5 := t2 + t4

p := t5

Consider the grammar.

(1) S → A a

(2) S → b A c

(3) S → d c

(4) S → b d a

(5) A → d

Parse input string "bdc" constructing LALR Parsing Table. Is the grammar LALR(1)?

Solution

The first number the production as below –

Step1– Construct Augmented Grammar

(0) S' → S

(1) S → A a

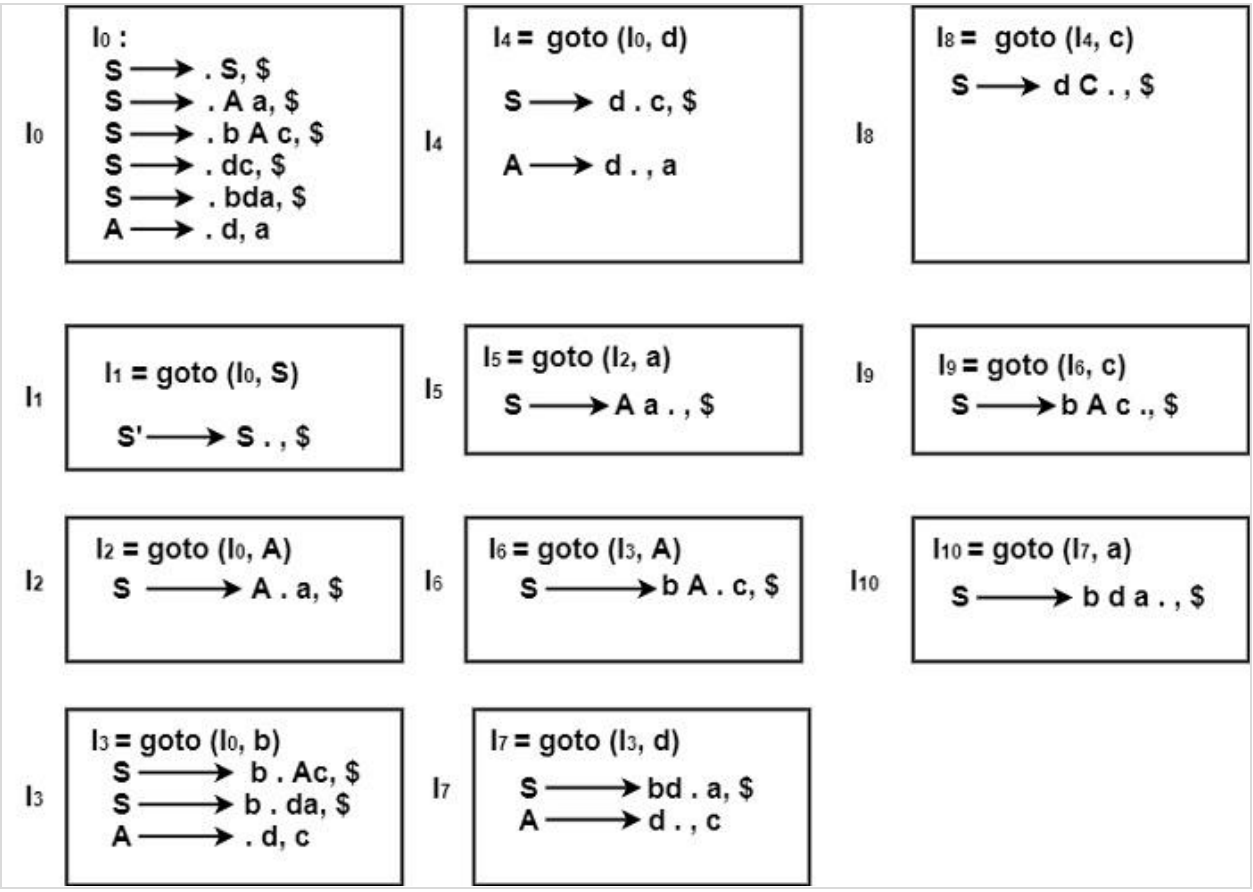
(2) S → b A c

(3) S → d c

(4) S → b d a

(5) A → d

Step2– Find Closure & goto. Find the canonical set of LR (1) items for the Grammar.



In the states, I0 to I10, no states have a similar first element or core. So, we cannot merge the states. Some states will be taken for building the LALR parsing table.

LALR Parsing Table

States		Action				goto	
	a	b	c	d	$	S	B
0		s3		s4		1	2
1					accept		
2	s5						
3				s7			6
4	r5		s8				
5							
6			s9				
7	s10		r5				
8					r3		
9					r2		
10					r4		

Parsing of String "bdc"

Stack	Input String	Action
$ 0	bdc $	Shift 3
$ 0 b 3	dc $	Shift 7
$ 0 b 3 d 7	c $	Reduce by A → d
$ 0 b 3 A 6	c $	Shift 9
$ 0 b 3 A 6 c 9	$	Reduce by S → b Ac

$ 0 s 1	$	accept

Consider the following grammar:
E-> E+T|T
T-> TF|F
F-> F*|a|b
Construct the SLR parsing table and also parse the input “a*b+a”

Solution

Step1 – Construct the augmented grammar and number the productions.

(0) E' → E

(1) E → E + T

(2) E → T

(3) T → TF

(4) T → F

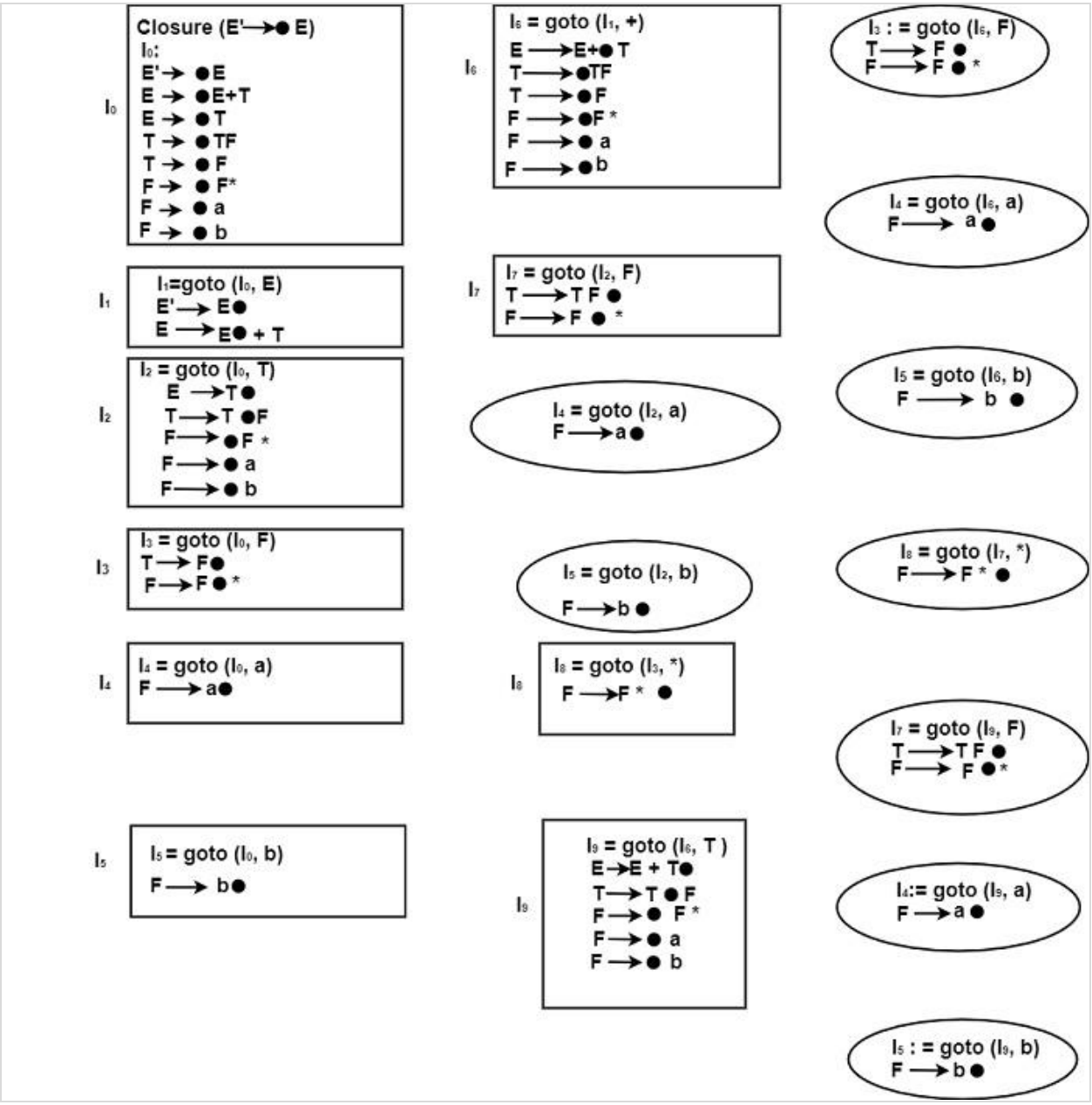
(5) F → F *

(6) F → a

(7) F → b.

Step2 – Find closure & goto Functions to construct LR (0) items.

Box represents the New states, and the circle represents the Repeating State.



Computation of FOLLOW

We can find out

FOLLOW(E) = {+, $}

FOLLOW(T) = {+, a, b, $}

FOLLOW(F) = {+,*, a, b, $}

LR Parsing Table

State	Action					goto		
	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				accept			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r6	r6	r6	r6	r6			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

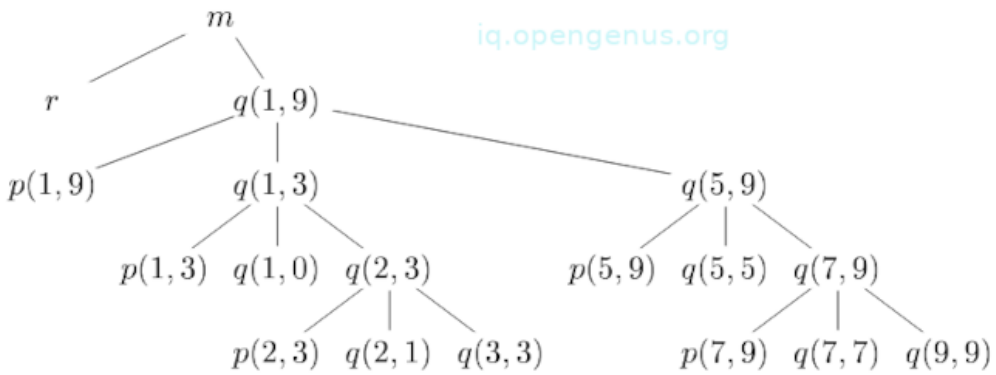
Parsing for Input String a * b + a –

Stack	Input String	Action
0	a * b + a $	Shift
0 a 4	* b + a $	Reduce by F → a.
0 F 3	* b + a $	Shift
0 F 3 * 8	b + a $	Reduce by F → F *
0 F 3	b + a $	Reduce by T → F
0 T 2	b + a $	Shift
0 T 2 b 5	+a $	Reduce by F → b
0 T 2 F 7	+a $	Reduce by T → TF
0 T 2	+a $	Reduce by E → T
0 E 1	+a $	Shift
0 E 1 + 6	a$	Shift
0 E 1 + 6 a 4	$	Reduce by F → a
0 E 1 + 6 F 3	$	Reduce by T → F
0 E 1 + 6 T 9	$	Reduce by E → E + T
0 E 1	$	Accept

Design an activation tree for the for the following function calls:
enter main()

enter readarray()
leave readarry()
enter quicksort(1,9)
    enter partition(1,9)
    leave partition(1,9)
    enter quicksort(1,3)
        …..
    leave quicksort(1,3)
    enter quicksort(5,9)
        …..

    leave quicksort(5,9)
    leave quicksort(1,9)
leave main()



Evaluate the execution state of the following C program

```
Main()
{
        int i;
        int a[10];
        i = 1;
        While(i <= 10)
        {
                a[i] = 0;
                i = i + 1;
        }
}
```

into

A. Syntax tree. B.Postfix notation. C.3 address code.

Consider the following grammar
S --> E
E --> E1 + T
E --> T
T --> T1 * F
T --> F
F --> digit
Draw the annotated parse tree to compute s-attribute for the input string 3 * 5 +5.

Efficient code generation requires the Remember of internal architecture of the target machine. Justify your answer with an Example

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

• It should carry the exact meaning of the source code.
• It should be efficient in terms of CPU usage and memory management.

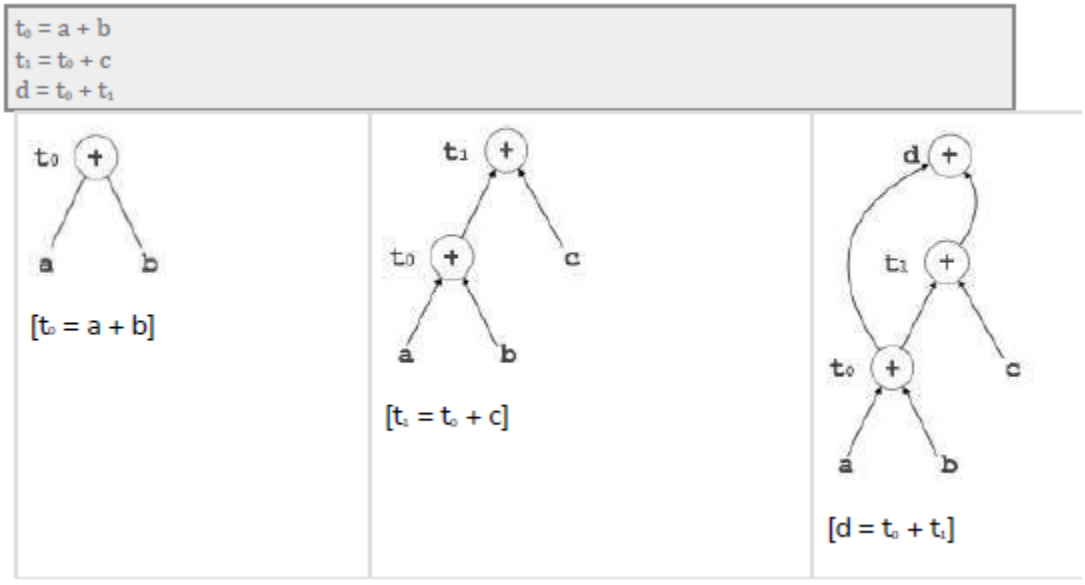
We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

• Leaf nodes represent identifiers, names or constants.
• Interior nodes represent operators.
• Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:



Flow of control optimization
There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code: ... MOV R1, R2 GOTO L1 ... L1 : GOTO L2 L2 : INC R1
In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below: ... MOV R1, R2 GOTO L2 ... L2 : INC R1

Code Generator
A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:
• Target language : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
• IR Type : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
• Selection of instruction : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.

Descriptors
The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:
• Register descriptor : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
• Address descriptor : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.
Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator: • updates the Register Descriptor R1 that has value of x and • updates the Address Descriptor (x) to show that one instance of x is in R1.
Code Generation
Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

getReg : Code generator uses getReg function to determine the status of available registers and the location of name values. getReg works as follows:
• If variable Y is already in register R, it uses that register.
• Else if some register R is available, it uses that register.
• Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

.