QA : Compiler Constructor

Created by	Ⓑ Borhan
Last edited time	@November 6, 2024 10:53 PM
Tag	

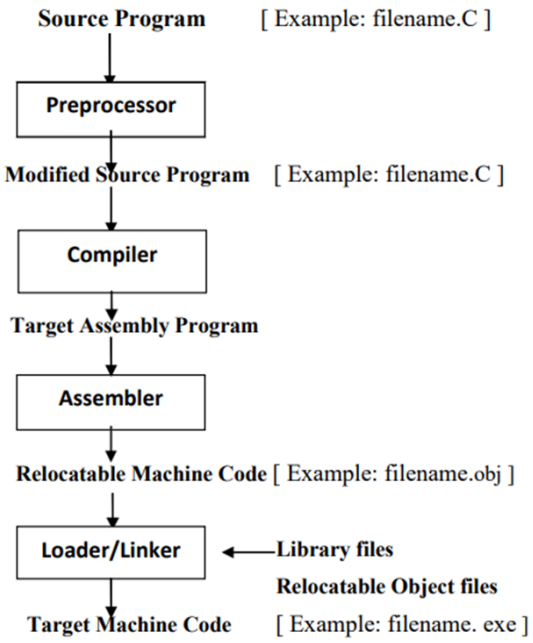
Resources

- Best: https://www.youtube.com/playlist?list=PLxCzCOWd7aiEKtKSIHYusizkESC42diyc
- **Operator Precedence Function :** https://www.youtube.com/watch?v=2sIHp7ny47o&ab_channel=WITSolapur-ProfessionalLearningCommunity
- https://www.tutorialspoint.com/construct-the-slr-parsing-table-for-the-following-grammar-also-parse-the-input-string-a-b-plus-a

Slide 1-7

- **Define language processing system? Briefly describe the phase of language processing system.**

The program which translate the program written in a programming language by the user into an executable program is known as language processors.



- It includes all header files and also evaluates whether a macro
- The compiler takes the modified code as input and produces the target code as output.
- The assembler takes the target code as input and produces real locatable machine code as output.
- Linker or link editor is a program that takes a collection of objects (created by assemblers and compilers) and combines them into an executable program.

- The loader keeps the linked program in the main memory.

- **Why it is necessary to divide compilation process in to various phases?**

Reason	Explanation
Modularity	Each phase handles a specific aspect of compilation, making the process easier to manage and understand. This separation allows for modifications in one phase without affecting others.
Error Detection	By breaking down the compilation into phases, errors can be detected and reported at different stages, facilitating easier debugging and correction of code.
Optimization	Each phase can apply specific optimizations relevant to that stage, improving overall performance without compromising the functionality of the code.
Separation of Concerns	Each phase addresses distinct tasks such as lexical analysis, syntax analysis, semantic analysis, optimization, and code generation, enhancing clarity and focus.
Efficiency	Phases can be designed to handle specific tasks in an efficient manner, allowing for parallel processing and better resource utilization.
Maintainability	A phased approach allows for easier updates and maintenance, as changes can be localized to specific phases without requiring a complete overhaul of the compiler.
Support for Different Languages	Phases can be tailored to support multiple programming languages by adjusting only the relevant parts of the compilation process, making the compiler more versatile.

- **Differentiate between context-free-grammar and regular expression.**

Aspect	Context-Free Grammar (CFG)	Regular Expression (RE)
Language Class	Context-Free Languages (CFL)	Regular Languages (RL)
Expressiveness	More powerful; handles nested structures	Limited; cannot handle nesting
Parsing Complexity	Requires complex parsers (e.g., LL, LR)	Parsed with finite automata (simpler)
Usage	Syntax of programming languages, nested structures	Pattern matching in text, simple validations
Grammar Structure	Defined by production rules in the form of `A → α`, where `A` is a non-terminal and `α` is a	Defined by a combination of literals, operators (e.g., `*`, `+`), and metacharacters.

Aspect	Context-Free Grammar (CFG)	Regular Expression (RE)
	sequence of terminals and/or non-terminals.	

	Context Free Grammar is formal grammar, the syntax or structure of a formal language can be described using context-free grammar (CFG), a type of formal grammar. The grammar has four tuples: (V,T,P,S).
	
	`V - It is the collection of variables or non-terminal symbols.`
	`T - It is a set of terminals.`
	`P - It is the production rules that consist of both terminals and non-terminals.`
	`S - It is the starting symbol.`
		Regular expressions are ***sequences of characters that define search patterns***, primarily used for string matching within texts.

- **Write down a regular expression for fractional number and identifier. Draw the -NFA for the regular expression (a|b)*bab.**

digits → $[0-9]$

letters → $[A-Z]|[a-z]$

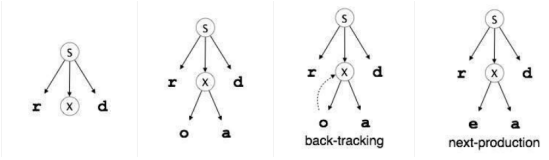
Fractional number → $(+|-)?digits^{+}(.digits^{+})?(E(+|-)?digits^{+})?$

Identifier → $(letters|\_)(letters|digits|\_)^{*}$

- **What is backtracking in top-down parsing?**

In Top-Down Parsing with Backtracking, Parser will attempt multiple rules or production to identify the match for input string by backtracking at every step of derivation. So, if the applied production does not give the input string as needed, or it does not match with the needed string, then it can undo that shift.

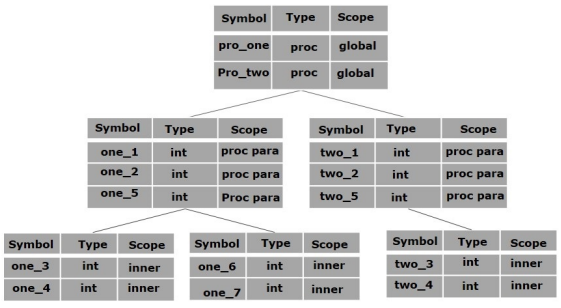
1. Start with the start symbol of the grammar
2. Apply the production rule for the non-terminal
3. Repeat step 2 in each step and check the input string
4. If mismatch, backtrack to previous step and apply the next alternative in production
5. Repeat step 4 until deriving the correct input string.



- **Shortly describe scope management of symbol table with proper example**

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.



- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

[https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm]

- **Define symbol table. Briefly describe four common error-recovery strategies that can be implemented in the parser to deal with errors in the code**

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.

1. **Panic Mode Recovery:** The parser skips tokens until it finds a synchronization point, like a semicolon or closing brace. This method prevents further errors by resuming parsing from a safe point, often leading to simpler and faster error recovery.
    - In this method, successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found. Synchronizing tokens are deli-meters such as ; or }
    - The advantage is that it's easy to implement and guarantees not to go into an infinite loop
    - The disadvantage is that a considerable amount of input is skipped without checking it for additional errors
2. **Phrase-Level Recovery**: The parser makes local corrections, such as inserting, deleting, or replacing tokens to continue parsing. This approach attempts to correct errors by modifying nearby tokens, allowing the parser to proceed with minimal disruption.
    - In this method, when a parser encounters an error, it performs the necessary correction on the remaining input so that the rest of the input statement allows the parser to parse ahead.
    - The correction can be deletion of extra semicolons, replacing the comma with semicolons, or inserting a missing semicolon.
    - While performing correction, utmost care should be taken for not going in an infinite loop.

    A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection
3. **Error Productions**: The parser includes specific grammar rules (error productions) to handle anticipated errors. When an error production matches an input pattern, the parser can give a more descriptive error message and proceed, aiding in the diagnosis of expected mistakes.

3. Error production

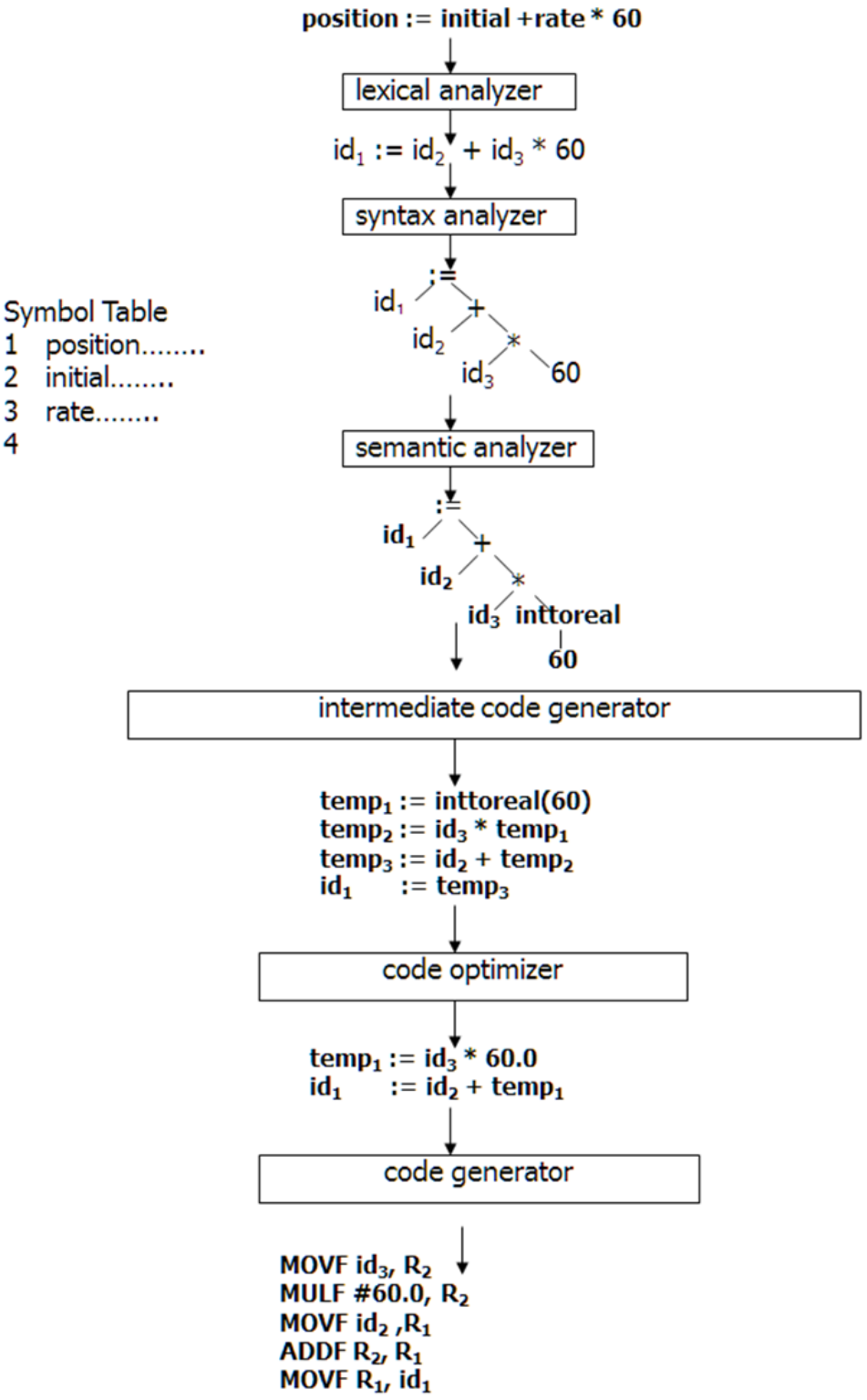
- ✓ If we have good knowledge of common errors that might be encountered, then we can augment the grammar for the corresponding language with error productions that generate the erroneous constructs.
- ✓ If error production is used during parsing, we can generate appropriate error message to indicate the erroneous construct that has been recognized in the input.
- ✓ This method is extremely difficult to maintain, because if we change grammar then it becomes necessary to change the corresponding productions.
- For Example: suppose the input string is **abcd**

Grammar:

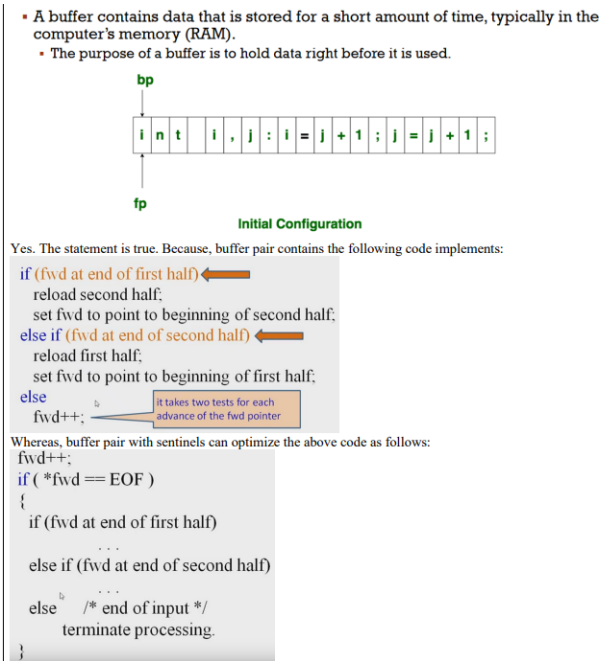
S→A
A→aA | bA | a | b
B→ cd

4. **Global Correction**: This approach involves finding the minimal set of changes to the input to make it syntactically correct, often using advanced algorithms. While it provides the best possible correction, it's computationally expensive and typically not used in real-time compilers
    - The parser examines the whole program and tries to find out the closest match for it which is error-free.
    - The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
    - Due to high time and space complexity, this method is not implemented practically

- **Show the phases of compiler for the statement position :=initial + rate*60**



- **What is input buffering? "Buffer pair with sentinels optimizes a code by reducing the number of tests"- do you agree with the statement? Justify your answer accordingly with an example.**



- **How to recover error using panic mode error recovery in LL(1) parser? Explain.**

It is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. The synchronizing set should be chosen so that the parser recovers quickly from error that are likely to occur in practice .

Sr. No.	Stack	Input	Remark
1	$E	)id*+id$	Error, skip)
2	$E	id*+id$	id is in FIRST(E)
3	$E'T	id*+id$	
4	$E'T'F	id*+id$	
5	$E'T'id	id*+id$	
6	$E'T'	*+id$	
7	$E'T'F*	*+id$	
8	$E'T'F	+id$	Error, M[F, +] = synch
9	$E'T'	+id$	F has been popped
10	$E'	+id$	
11	$E'T+	+id$	
12	$E'T	id$	
13	$E'T'F	id$	
14	$E'T'id	id$	
15	$E'T'	$	
16	$E'	$	
17	$	$	

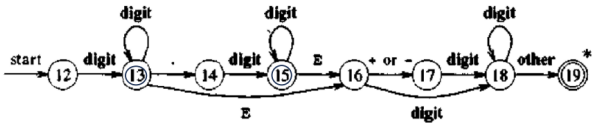
E→TE' E' →+TE'|ε
T →FT' T' →*FT'|ε
F →(E)|id

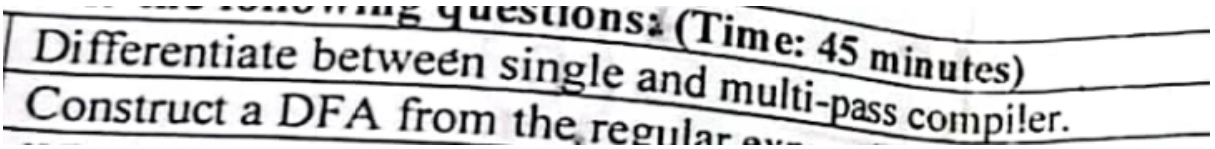
Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'	*synch*	*synch*
E'		E' →+TE'			E' →ε	E' →ε
T	T →FT'	*synch*		T →FT'	*synch*	*synch*
T'		T' →ε	T' →*FT'		T' →ε	T' →ε
F	F →id	*synch*	*synch*	F →(E)	*synch*	*synch*

- If the parser looks up the entry M[A, a] and finds that it is blank, then the input symbol a is skipped.
- If the entry is synch, then the nonterminal on the top of the stack is popped in an attempt to resume parsing.

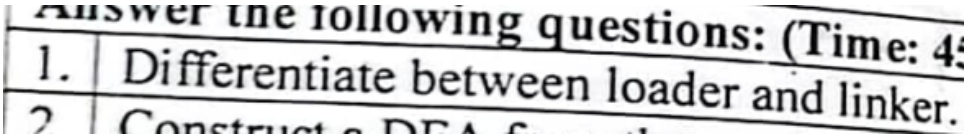
- If a token on the top of the stack does not match the input symbol, then we pop the token from the stack.

- **Draw a transition diagram accepting both integer and floating-point numbers with exponentiation.**

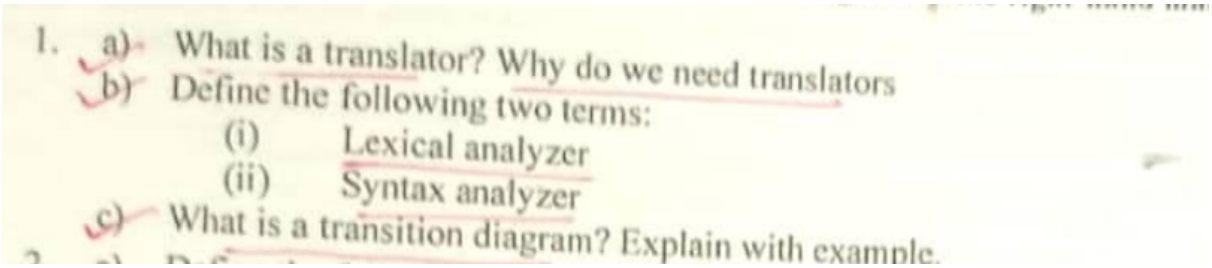




One Pass Compiler	Two Pass Compiler/Multi pass
It performs Translation in one pass	It performs Translation in two pass
It scans the entire file only once.	It requires two passes to scan the source file.
It doesn’t generate intermediate code	It generates intermediate code
Speed fast	Speed slow
Time less	Time more
Memory more	memory less
not portable	portable



S. No.	Linker	Loader
1	A linker is an important utility program that takes the object files, produced by the assembler and compiler, and other code to join them into a single executable file.	A loader is a vital component of an operating system that is accountable for loading programs and libraries.
2	It uses an input of object code produced by the assembler and compiler.	It uses an input of executable files produced by the linker.
3	The foremost purpose of a linker is to produce executable files.	The foremost purpose of a loader is to load executable files to memory.
4	Linker is used to join all the modules.	Loader is used to allocate the address to executable files.
5	It is accountable for managing objects in the program's space.	It is accountable for setting up references that are utilized in the program.


1. a) What is a translator? Why do we need translators
b) Define the following two terms:
(i) Lexical analyzer
(ii) Syntax analyzer
c) What is a transition diagram? Explain with example.

(a) Translator is a program that converts code written in one language into another language. This process is essential in computing because it allows programs written in high-level, human-readable languages to be transformed into low-level machine code, which the computer can execute.

(b)

The **lexical analyzer**, also known as a **scanner** or **lexer**, is the first phase of a compiler. It reads the source code character by character and groups these characters into meaningful sequences called **tokens**.

The
syntax analyzer, also known as the **parser**, is the second phase of a compiler. It

takes the tokens produced by the lexical analyzer and arranges them into a tree-like structure called a **parse tree** or **syntax tree**

(c)

Transition diagram can be interpreted as a flowchart for an algorithm recognizing a language.

Which is constructed by

- There is a node for each state in Q, represented by the circle
- There is a directed edge from a node p to node q labeled if $\delta(p, a) = p$
- Starting state, there is an arrow with no source
- Final states indicating by a double circle

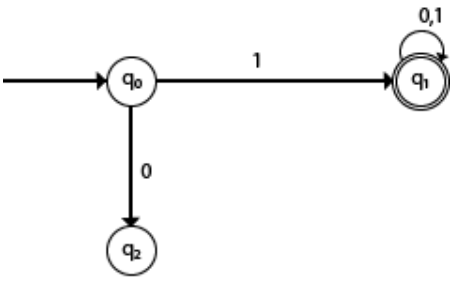


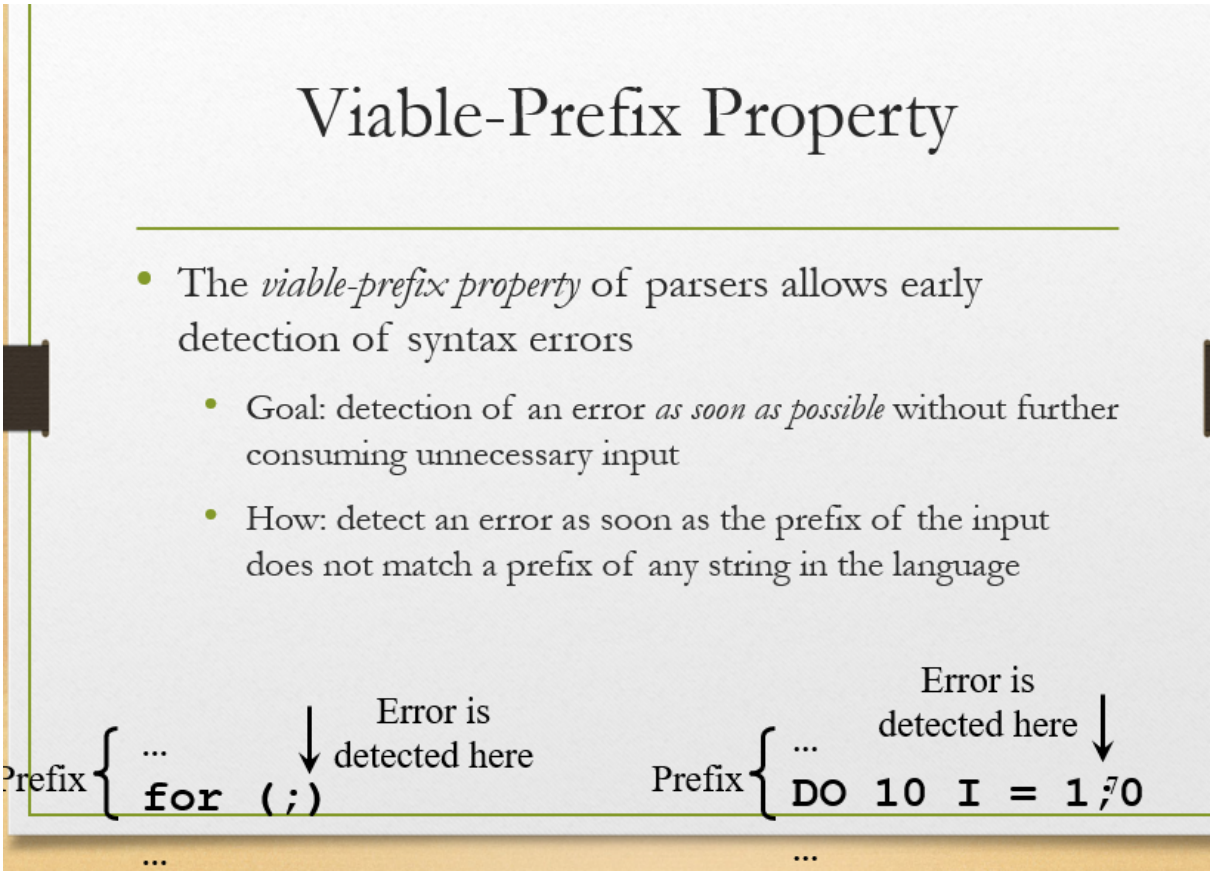
Fig: Transition diagram

b) Differentiate among token, pattern and lexeme with examples.

A token is a pai**r consisting of a token name and an optional attribute value.**
A pattern is a description of the
form that the lexemes of a token may take [or match].
A lexeme is a
sequence of characters in the source program that **matches the pattern for a token** and is identified by the lexical analyzer as an instance of that token.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but ", surrounded by "'s	`"core dumped"`

c) What is viable prefix property? Explain with an example.



a) Write some common examples of syntactic errors.

- Error in structure
- Missing operators
- Unbalanced parenthesis

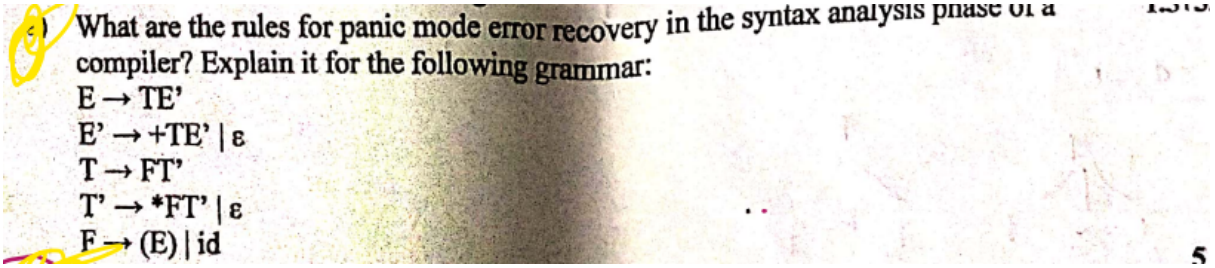
```
int a = 5          // semicolon is missing
x = (3 + 5;  // missing closing parenthesis ')'
y = 3 + * 5;   // missing argument between '+' and '*'
```

(b)What are the points in the parsing process at which an operator-precedence parser can discover syntactic errors?

There are two points in the parsing process at which an operator-precedence parser can discover syntactic error:

- If no precedence relation holds between the terminal on top of the stack and the current input.

- If a handle has been found, but there is no production with this handle as a right side.



What are the rules for panic mode error recovery in the syntax analysis phase of a compiler? Explain it for the following grammar:

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

5

✓ **Panic mode recovery**:

Rules:

1. If parser looks up entry *M[A, a]* and finds it blank then the input symbol *a* is skipped.
2. If the entry is *synch*, then the nonterminal on top of the stack is popped in an attempt to resume parsing.
3. If a token on the top of the stack does not match the input symbol, then we pop the token from the stack

Sr. No.	Stack	Input	Remark
1	$E	)id*+id$	Error, skip)
2	$E	id*+id$	id is in FIRST(E)
3	$E'T	id*+id$	
4	$E'T'F	id*+id$	
5	$E'T'id	id*+id$	
6	$E'T'	*+id$	
7	$E'T'F*	*+id$	
8	$E'T'F	+id$	Error, M[F, +] = synch
9	$E'T'	+id$	F has been popped
10	$E'	+id$	
11	$E'T+	+id$	
12	$E'T	id$	
13	$E'T'F	id$	
14	$E'T'id	id$	
15	$E'T'	$	
16	$E'	$	
17	$	$	

E→TE' E' →+TE'|ε
T →FT' T' →*FT'|ε
F →(E)|id

Non-terminal	Input Symbol					
	id	+	*	(	)	$
E	E→TE'			E→TE'	*synch*	*synch*
E'		E' →+TE'			E' →ε	E' →ε
T	T →FT'	*synch*		T →FT'	*synch*	*synch*
T'		T' →ε	T' →*FT'		T' →ε	T' →ε
F	F →id	*synch*	*synch*	F →(E)	*synch*	*synch*

- **Show the comparisons among error recovery strategies in a lexical analyzer with examples.**

Error Recovery Strategy	Description	Example	Pros	Cons
Panic Mode	Skips tokens until it reaches a known, predefined delimiter (e.g., a semicolon `;` or end of statement) to resume processing from a safe point.	If an error occurs in `int x = 10 y;`, the analyzer skips `y;` and resumes after `;`.	Simple and quick recovery; avoids cascading errors.	Loss of tokens; may skip over valid code unintentionally.
Error Token Insertion	Inserts an artificial token to make the input stream valid and continue processing.	In `int x = ;`, the lexer may insert a placeholder token (e.g., `0`) after `=` to continue.	Preserves structure; often used for minor syntax errors.	Can lead to inaccuracies by adding tokens that aren't part of the original input.
Error Token Deletion	Deletes offending tokens to resolve the error and move forward.	In `int x 10 = 20;`, deleting `10` can allow `x = 20;` to be parsed correctly.	Efficient in certain contexts; reduces clutter by removing extraneous tokens.	Risks deleting too many tokens; may alter the intended meaning of the code.
Transpose two serial characters				
Replace a character with another character				

- **What do you mean by an ambiguous grammar? What are the main reasons of ambiguity and how can ambiguity be eliminated?**

Ambiguous grammar is a grammar which produces more than one parse tree for a same string.

The main reasons:

- Precedence
- Associativity

- Dangling else

Remove:

- Rewriting the grammar
- Use ambiguous grammar with additional rules

b) Consider the following ambiguous grammar:

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

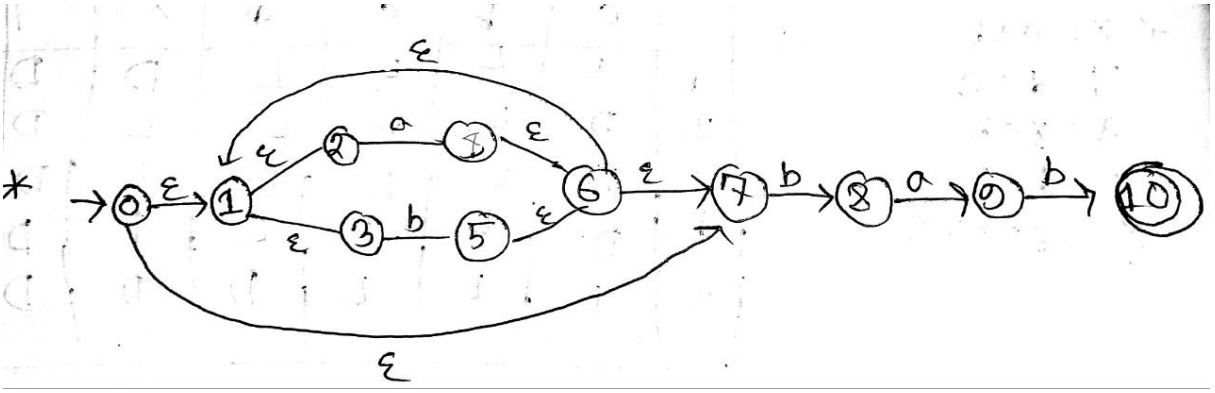
Rewrite the above grammar to eliminate ambiguity and then show the derivation for the compound conditional statement "if E1 then S1 else if E2 then S2 else S3".

- Rewrite the dangling else grammar as unambiguous grammar
- The grammar will be

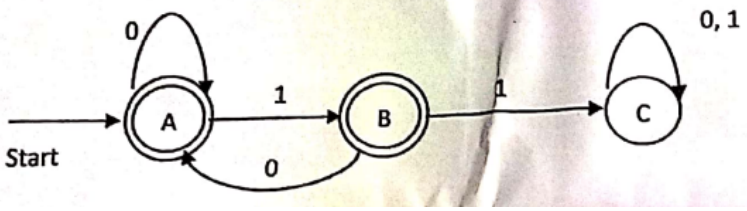
Stmt → *matchedstmt*
| openstmt
***matchedstmt* → if expr then matchedstmt else matchedstmt**
| other
openstmt → if expr then stmt
| if expr then matchedstmt else openstmt

If E1 then if E2 then S1 else S2

b) Draw an NFA for the regular expression **(a|b)*abb** using ε-transition and construct an equivalent DFA for the expression.



Consider the following figure, find the language of it.



What is left recursion of a grammar? Consider the following grammar:

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

Eliminate left recursion from the grammar.

* $i = 1$, nothing to do

$i = 2$, $j = 1$: $A \rightarrow Ac \mid Sd \mid \varepsilon$

$\Rightarrow A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

$\Rightarrow A \rightarrow bdA' \mid \varepsilon$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Another one to remove Left recursion

$E \rightarrow E+T \mid T$

$\Rightarrow E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow T*F \mid F$

$\Rightarrow T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

What is left recursion of a grammar? Eliminate left recursion from the following grammar:

$A \rightarrow B\ C \mid a$

$B \rightarrow C\ A \mid A\ b$

$C \rightarrow A\ B \mid C\ C \mid a$

Answer:

A Grammar G is left recursive Grammar if the non-terminal A in the derivation is of the form:

$$\mathbf{A} \overset{+}{\Rightarrow} \mathbf{A}\alpha$$

Where α is a string of terminals and non-terminals.

Whenever the first symbol in the **right hand side of the production is same as the left hand side variable**, then the grammar is said to be a **left recursive grammar**.

$i = 1$: nothing to do

$i = 2, j = 1$: $B \rightarrow C\ A \mid \underline{A}\ \mathbf{b}$

$\Rightarrow B \rightarrow C\ A \mid \underline{B\ C}\ \mathbf{b} \mid \underline{\mathbf{a}}\ \mathbf{b}$

$\Rightarrow_{(imm)} B \rightarrow C\ A\ B_R \mid \mathbf{a}\ \mathbf{b}\ B_R$

$B_R \rightarrow C\ \mathbf{b}\ B_R \mid \varepsilon$

$i = 3, j = 1$: $C \rightarrow \underline{A}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow C \rightarrow \underline{B\ C}\ B \mid \underline{\mathbf{a}}\ B \mid C\ C \mid \mathbf{a}$

$i = 3, j = 2$: $C \rightarrow \underline{B}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow C \rightarrow \underline{C\ A\ B_R}\ C\ B \mid \underline{\mathbf{a}\ \mathbf{b}\ B_R}\ C\ B \mid \mathbf{a}\ B \mid C\ C \mid \mathbf{a}$

$\Rightarrow_{(imm)} C \rightarrow \mathbf{a}\ \mathbf{b}\ B_R\ C\ B\ C_R \mid \mathbf{a}\ B\ C_R \mid \mathbf{a}\ C_R$

$C_R \rightarrow A\ B_R\ C\ B\ C_R \mid C\ C_R \mid \varepsilon$

4.	Prove that the following grammar is not LL(1): $S \rightarrow \mathbf{i}E\mathbf{t}SS' \mid \mathbf{a}$ $S' \rightarrow \mathbf{e}S \mid \varepsilon$ $E \rightarrow \mathbf{b}$

Ambiguous grammar

$S \rightarrow \mathbf{i}\, E\, \mathbf{t}\, S\, S_R \mid \mathbf{a}$

$S_R \rightarrow \mathbf{e}\, S \mid \varepsilon$

$E \rightarrow \mathbf{b}$

$A \rightarrow \alpha$	FIRST(α)	FOLLOW(A)
$S \rightarrow \mathbf{i}\, E\, \mathbf{t}\, S\, S_R$	i	e $
$S \rightarrow \mathbf{a}$	a	
$S_R \rightarrow \mathbf{e}\, S$	e	e $
$S_R \rightarrow \varepsilon$	ε	
$E \rightarrow \mathbf{b}$	b	t

Error: duplicate table entry

	a	b	e	i	t	$
S	$S \rightarrow \mathbf{a}$			$S \rightarrow \mathbf{i}\, E\, \mathbf{t}\, S\, S_R$		
S_R			$S_R \rightarrow \varepsilon$ $S_R \rightarrow \mathbf{e}\, S$			$S_R \rightarrow \varepsilon$
E		$E \rightarrow \mathbf{b}$				

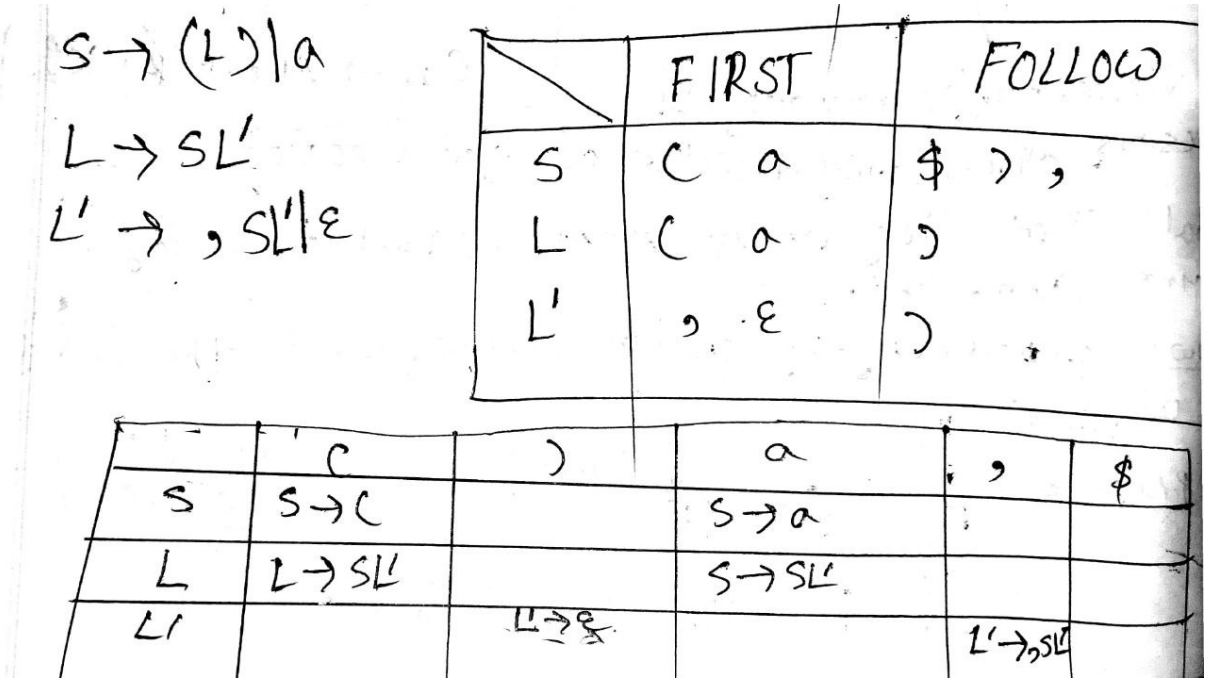
Calculate FIRST and Follow for the following grammar:

$S \rightarrow (L) \mid \mathbf{a}$

$L \rightarrow L, S \mid S$

(With check LL(1) or not)

[Used space as separator in FIRST and FOLLOW]



4. What are the conditions of LL(1) grammar? Prove that the following grammar is whether LL(1) or not:
$S \rightarrow CC \mid \mathbf{a}$
$C \rightarrow cC \mid d$

Answer:
Part 1:

- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - ✓ **Condition 1**: For no terminal a, do both α and β derive strings beginning with a. (FIRST(α) ∩ FIRST(β) = ϕ)
  - ✓ **Condition 2**: At most one of α and β can derive empty string.
  - ✓ **Condition 3**: If $\beta \xrightarrow{*} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A). (FIRST(α) ∩ FOLLOW(A) = ϕ)

S → CC | a

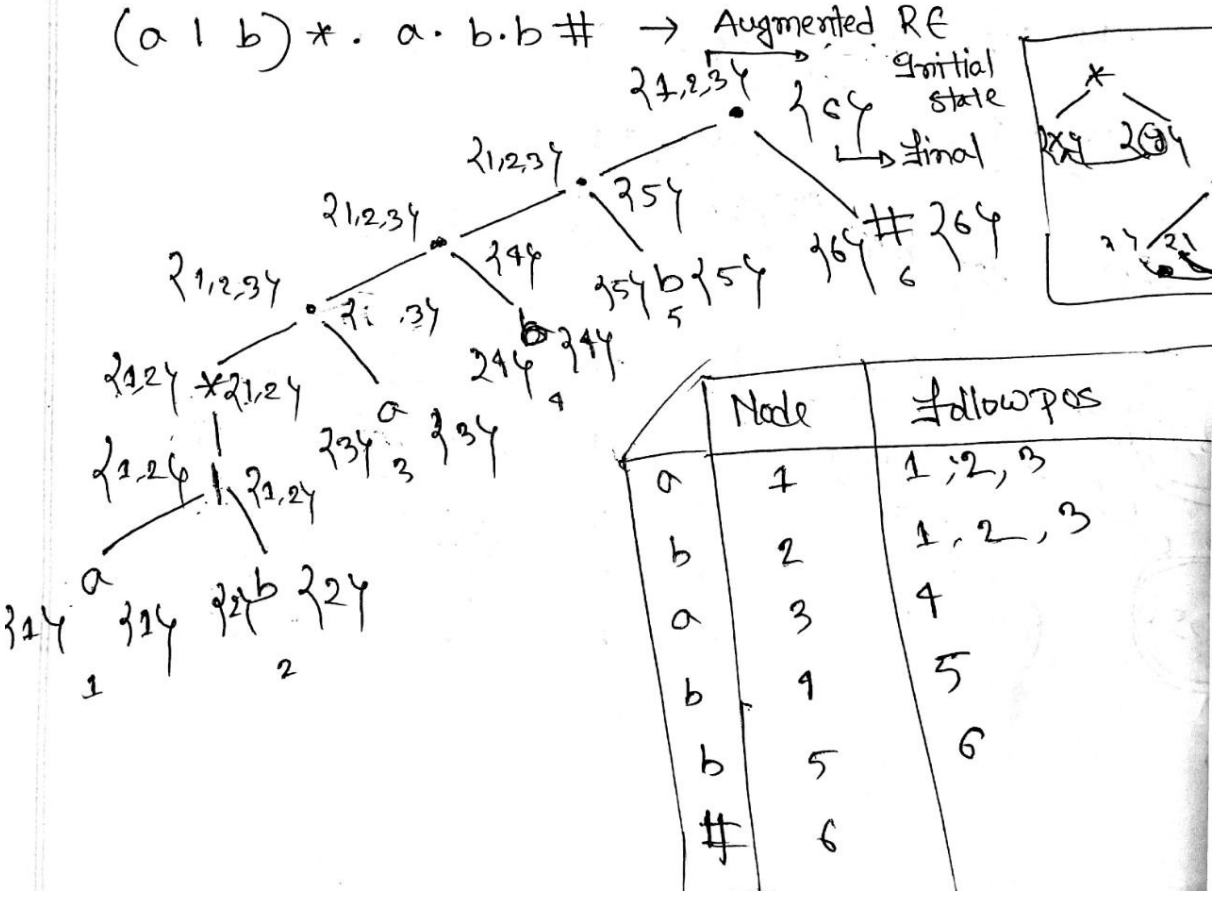
C → cC | d

	First	Follow
S	c, a, d	$
C	c, d	c, d, $

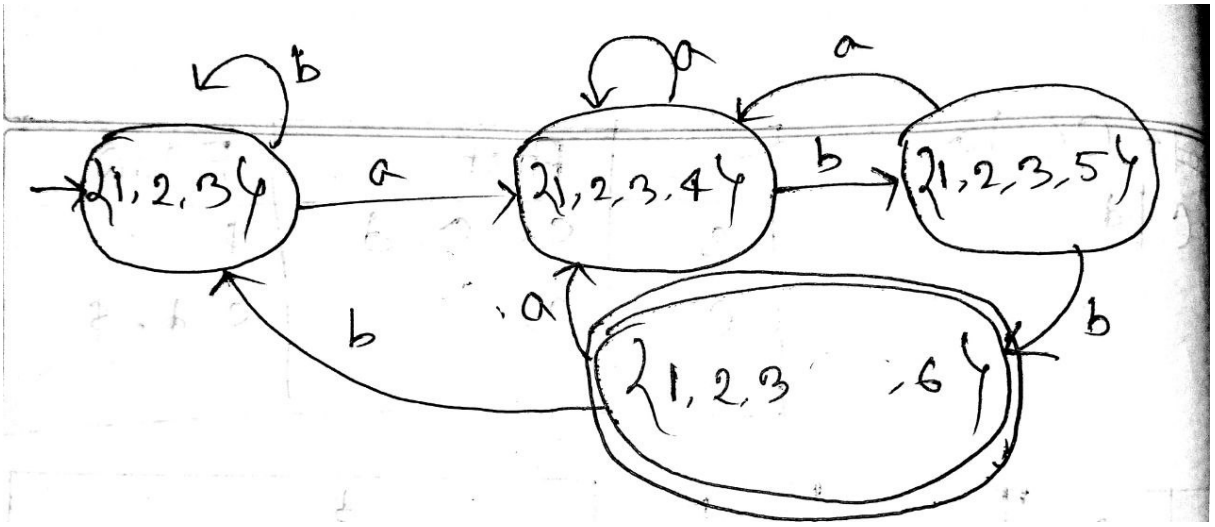
	c	d	a	$
S	S→CC	S→CC	S→a	
C	C→cC	C→d		

2. Construct a DFA from the regular expression (a|b)*abb using direct method.
3. What is left recursion of a grammar?

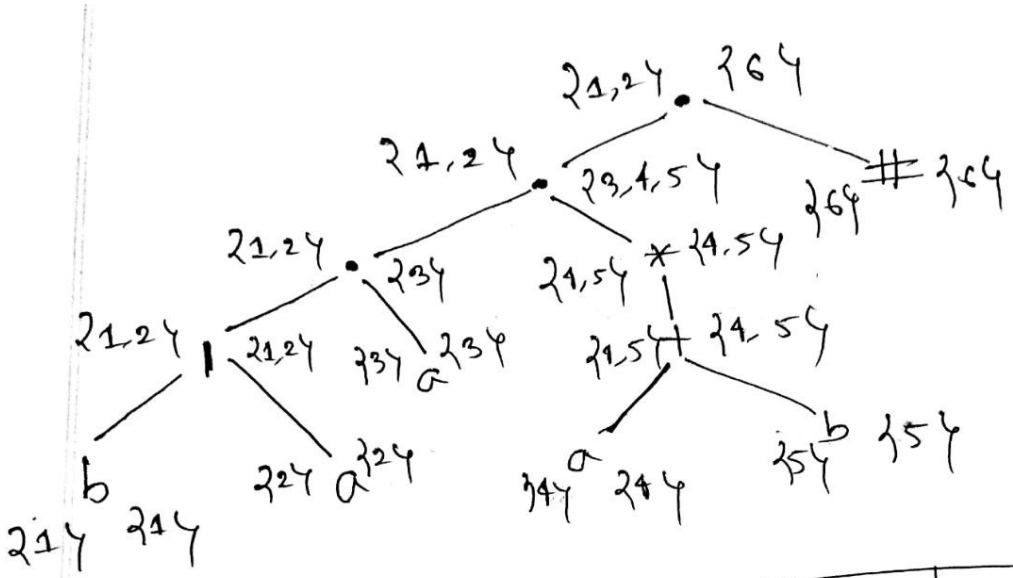
(a | b) * . a · b · b # → Augmented RE

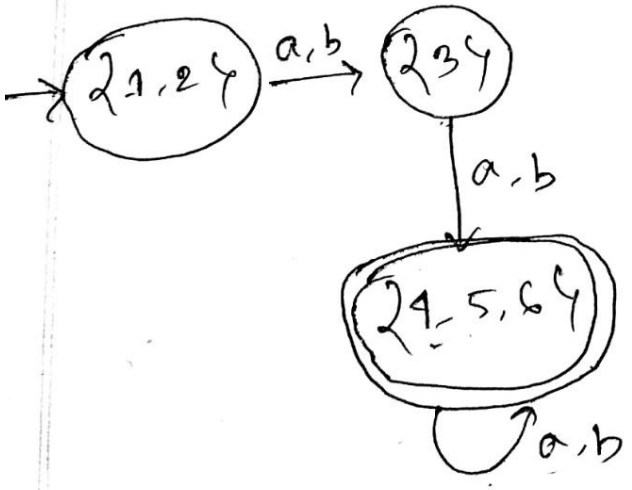


	Node	Followpos
a	1	1, 2, 3
b	2	1, 2, 3
a	3	4
b	4	5
b	5	6
#	6	



	Differentiate between loader and linker.
2.	Construct a DFA from the regular expression (b|a)a(a+b)* using direct method.
3.	Explain about recursive de...





	Node	Followpos
b	1	3
a	2	3
a	3	4, 5, 6
a	4	4, 5, 6
b	5	4, 5, 6
#	6	

Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Slide 7-14

Develop an algorithm for constructing an SLR parsing table. Write down the phases of a compiler.

Input : An augmented grammar G′

Output : The SLR parsing table functions action and goto for G′

Method :

1. Construct C = {$I_0, I_1, \ldots I_n$}, the collection of sets of LR(0) items for G'.
2. State *i* is constructed from I_i. The parsing functions for state *i* are determined as follows:
   (a) If [A→α·*a*β] is in I_i and goto(I_i,*a*) = I_j, then set *action*[*i*,*a*] to "shift j". Here *a* must be terminal.
   (b) If [A→α·] is in I_i, then set *action*[*i*,*a*] to "reduce A→α" for all *a* in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set *action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule: If *goto*(I_i,A) = I_j, then *goto*[i,A] = *j*.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the set of items containing [S'→.S].

Phases of a compiler

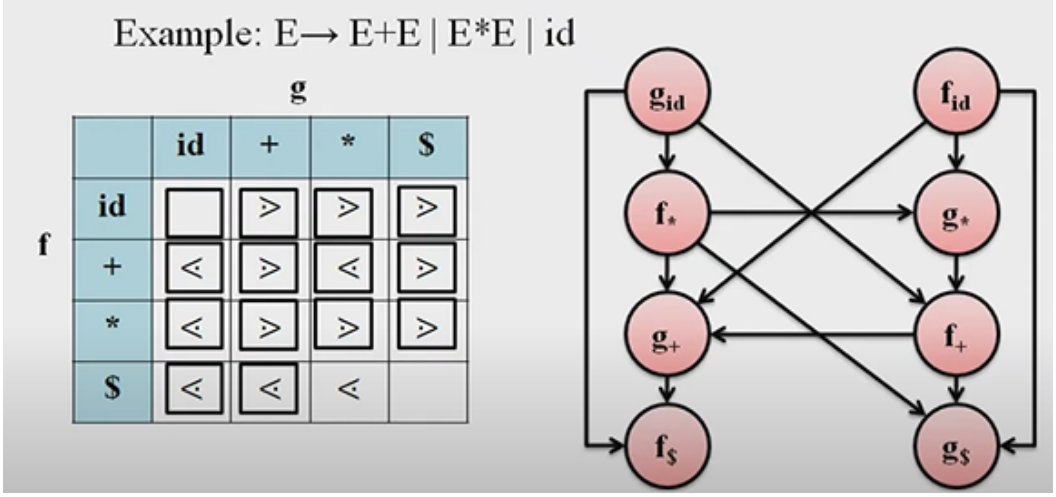
- Lexical Analysis
- Syntax Analysis
- Semantics Analysis
- Intermediate Code Generation
- Code Optimization
- Target Code Generation

What is precedence function? Show how to construct precedence function with an example.

P**recedence functions** that map terminal symbols to integers.

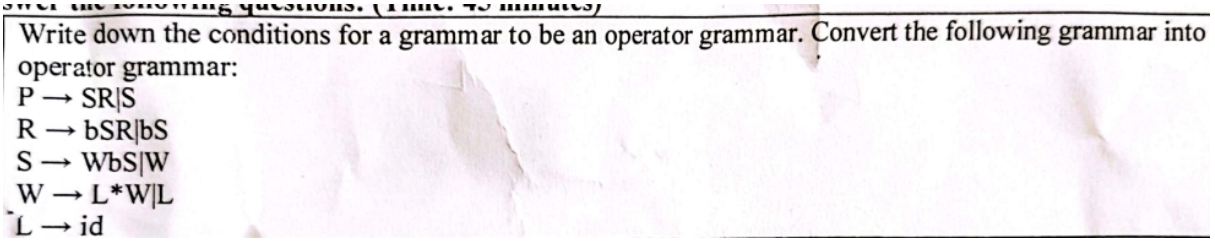
Precedence relations between any two operators or symbols in the precedence table can be converted to two precedence functions f & g that map terminals symbols to integers.

- For each terminal a, create this symbol fa and ga
- if a<. b, mark edge from gb to fa
- if a .> b, Mark an edge from fa to gb



Maximum Length for each Node

	Id	+	*	$
F	4	2	4	0
G	5	1	3	0


Write down the conditions for a grammar to be an operator grammar. Convert the following grammar into operator grammar:
P ⟶ SR|S
R ⟶ bSR|bS
S ⟶ WbS|W
W ⟶ L*W|L
L ⟶ id

Solution:

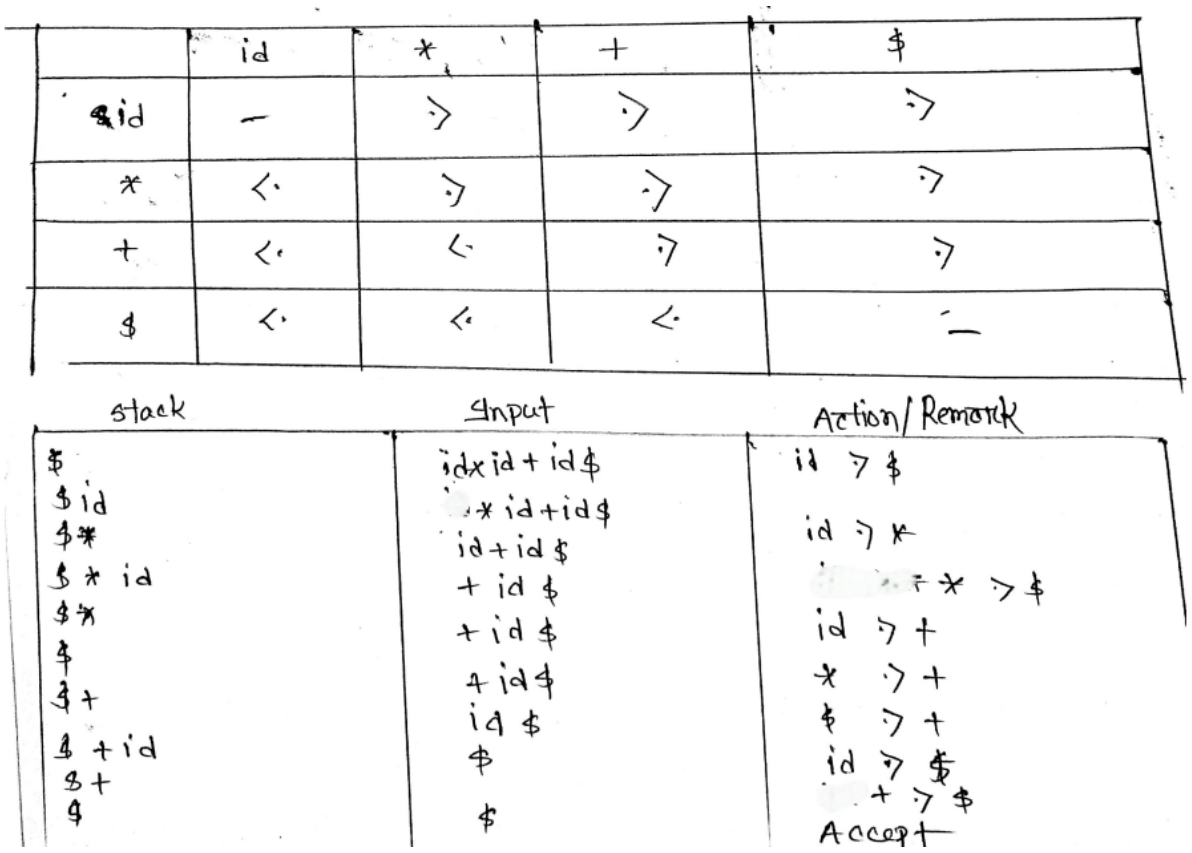
Conditions for a grammar to be an operator grammar :

- No R,H,S of any production has a ϵ
- No two non-terminals are adjacent.

$P \rightarrow SbP|SbS|S$
$R \rightarrow bP|bS$
$S \rightarrow WbS|W$
$W \rightarrow L*W|L$
$L \rightarrow id$

We can remove R, because it is unreachable.

Show the parsing steps for the string id * id+id using operator precedence parsing technique (Note: You must include the precedence table in your answer).



Another Question

	id	+	*	$
id	-	.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	-

STACK	INPUT	ACTION/REMARK
$	id + id * id$	$ <. Id
$ id	+ id * id$	id >. +
$	+ id * id$	$ <. +
$ +	id * id$	+ <. Id
$ + id	* id$	id .> *
$ +	* id$	+ <. *
$ + *	id$	* <. Id
$ + * id	$	id .> $
$ + *	$	* .> $
$ +	$	+ .> $
$	$	accept

c) The following grammar is operator-precedence and SLR (1). 5

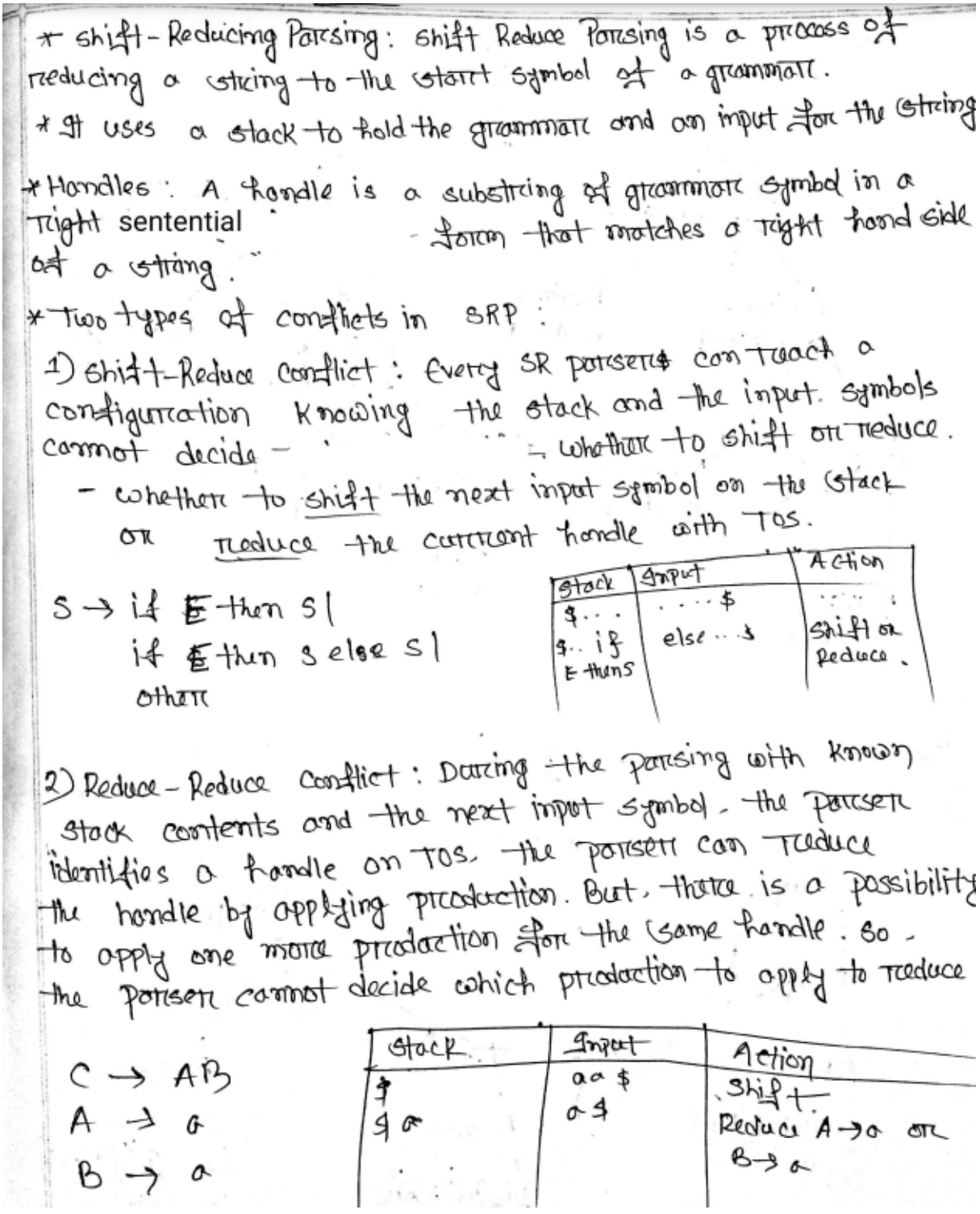
S→ **if** e **then** S **else**
| **while** e **do** S
| **begin** L **end**
| s

L→ S; L | S

Construct error-correcting parsers of the operator-precedence and LR type for this grammar.

- **What do you mean by Shift-Reduce conflict? Explain with an example.**

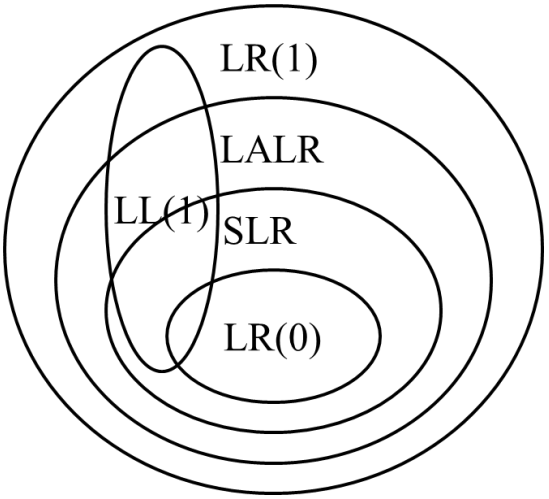
- **What do you mean by Reduce-Reduce conflict? Explain with an example.**



- **Differentiate among LR parsers.**

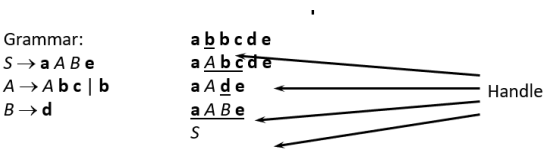
SLR Parser	LALR Parser	CLR Parser
Simple LR	Lookahead LR	Canonical LR

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 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.



- **Define handle with an example. What are the rules for constructing closure of item sets and goto operation?**

A handle is a substring of grammar symbol in a right-sentential form that matches a right hand side of a string.



If 'I' is a set of items for a grammar G then closure of I is set of items constructed I by two rules:

- Initially, add every item in I to colure (I)

Goto Operation

- If $A \to \alpha \cdot B\beta$ is in closure (I) and $B \to \gamma$ is a production, then add item $B \to \cdot\gamma$ to I, if it is not already in existence.
- Apply this rule until no more new items can be added to closure (I)

- If there is a production $A \to \alpha \cdot X\beta$ then $goto(A \to \alpha \cdot X\beta, X) = A \to \alpha X \cdot \beta$
- Simply shifting of dot (.) one position ahead over the grammar symbol.
- The rule $A \to \alpha \cdot X\beta$ is in I then the same goto function can be written as goto(I, B).
- Goto (I, X), where I is a set of items and X is a grammar symbol, is defined as the closure of the set of all items $[A \to \alpha X \cdot \beta]$ such that $[A \to \alpha \cdot X\beta]$ is in I.

- Example: If I is the set of two items {E' → E.], [E → E.+T]}, then goto(I, +) consists of

E → E + .T
T → .T * F
T → .F
F → .(E)
F → .id

Show how to parse the input *a * b + a* using the following grammar and the parsing table:

E → E+T | T
T → TF | F
F → 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

$E \to E+T$ (a)
$E \to T$ (2)
$T \to TF$ (3)
$T \to F$ (4)
$F \to F*$ (5)
$F \to a$ (6)
$F \to b$ (7)

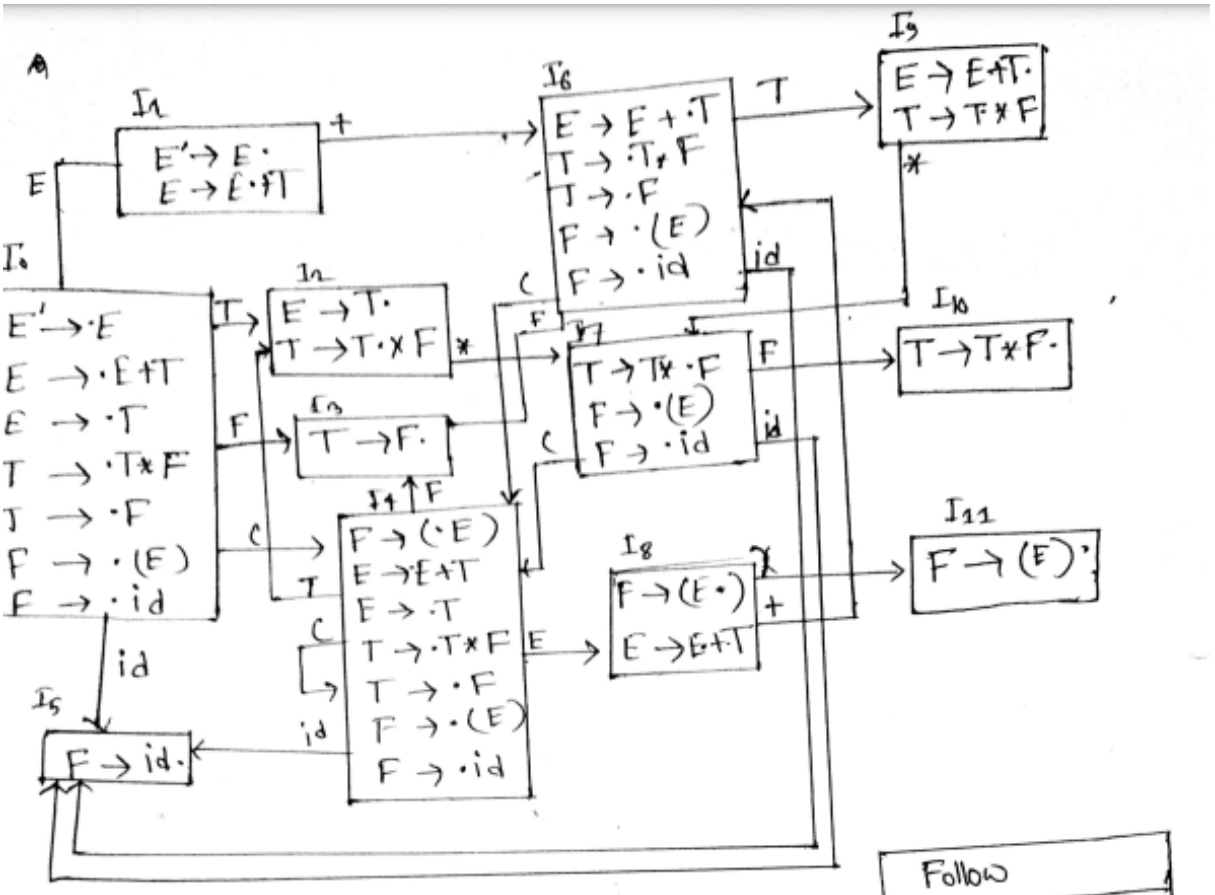
STACK	INPUT	ACTION
(1) 0	a*b+b$	shift
(2) 0 a 4	*b+b$	Reduced by F→a
(3) 0 F 3	*b+b$	shift
(4) 0 F 3 * 8	b+b$	Reduced by F→F*
(5) 0 F 3	b+b$	Reduced by T→F
(6) 0 T 2	b+b$	shift
(7) 0 T 2 b 5	+b$	Reduce by F→a
(8) 0 T 2 F 7	+b$	Reduced by T→TF
(9) 0 T 2	+b$	Reduced by E→T
(10) 0 E 1	+b$	shift
(11) 0 E 1 + 6	b$	shift
(12) 0 E 1 + 6 b 5	$	Reduced by F→a
(13) 0 E 1 + 6 F 3	$	Reduced by T→F
(14) 0 E 1 + 6 T 9	$	Reduced by E→E+T
(15) 0 E 1	$	Accept

Construct SLR parsing table for the following grammar:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Accept			
2		π_2	S_7		π_2	π_2			
3		π_4	π_4		π_4	π_4			
4	S_5			S_4			8	2	3
5		π_6	π_6		π_6	π_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		π_1	S_7		π_1	π_1			
10		π_3	π_3		π_3	π_3			
11		π_5	π_5		π_5	π_5			

Follow
E : {+,), $}
T : {+,), $, * }
F : {+,), $, * }

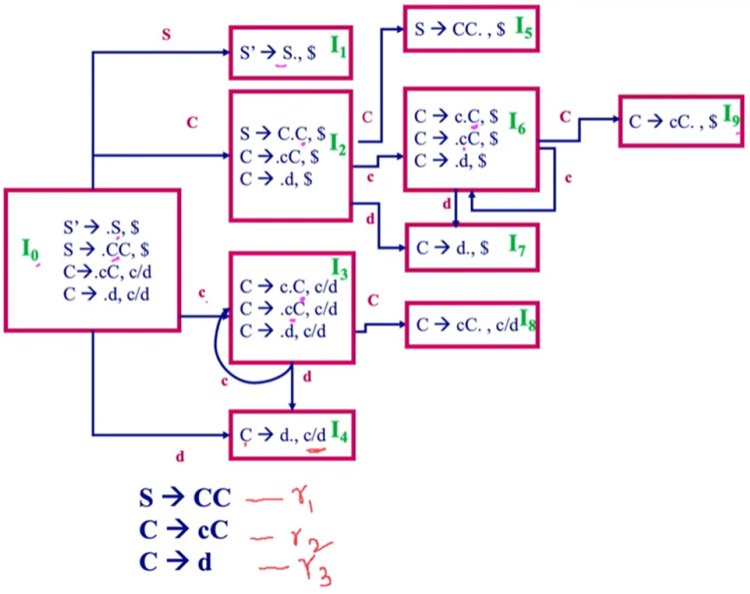
$E \to E+T$ – (1)
$E \to T$ – (2)
$T \to T*F$ – (3)
$T \to F$ – (4)
$F \to (E)$ – (5)
$F \to id$ – (6)

Construct CLR parsing table for the following grammar:

S → CC

C → cC

C → d



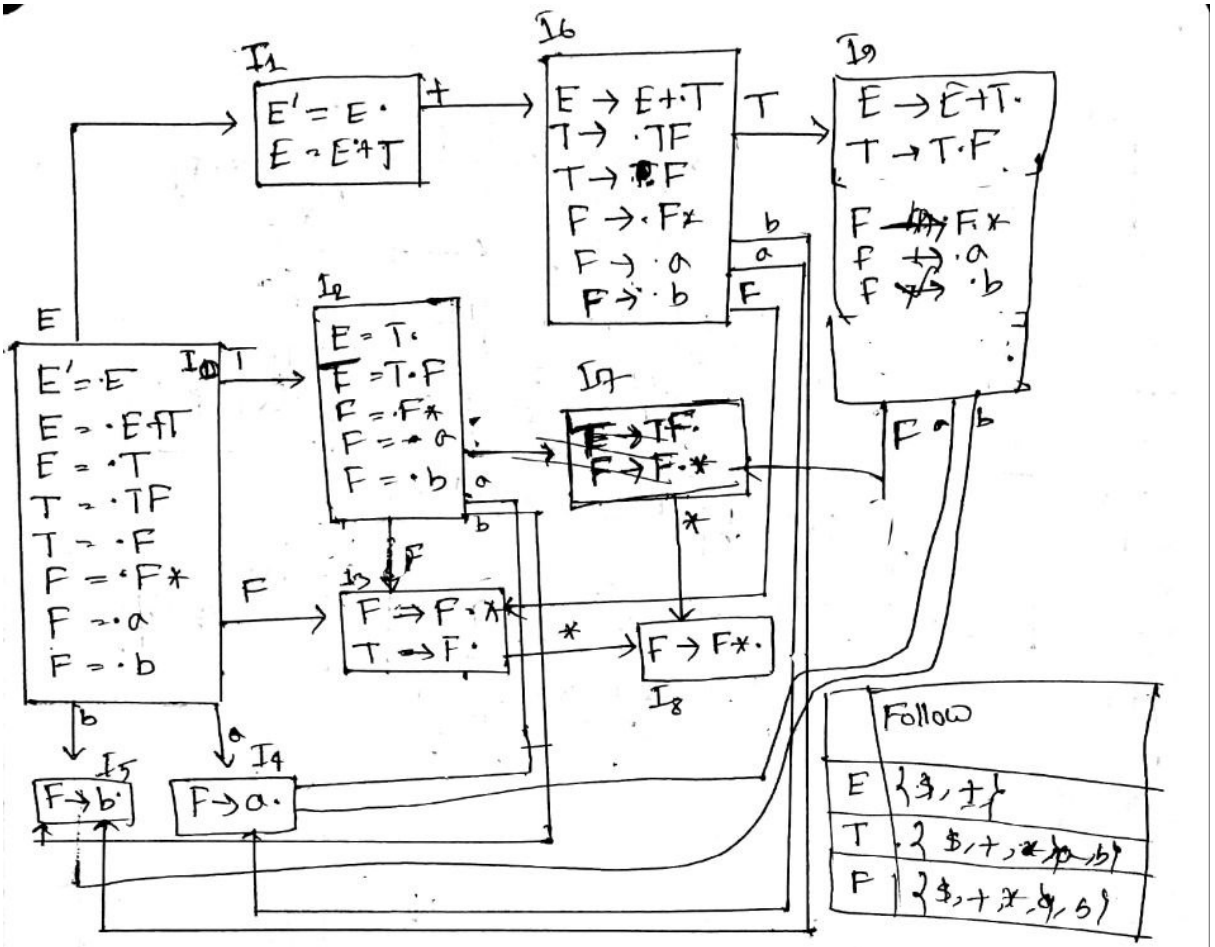
STATE	ACTION			GOTO	
	c	d	$	S	C
0	S_3	S_4		1	2
1			accept		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

Construct SLR parsing table for the following grammar:

E → E+T | T — 1

T → TF | F — 2

F → F* | a | b — 3



STATE	ACTION					GOTO		
	+	*	a	b	$	E	T	F
0			S_4	S_5		1	2	3
1	S_6				Accept			
2	π_2		S_4	S_5	π_2			7
3	π_4	S_8	π_4	π_4	π_4			
4	π_6	π_6	π_6	π_6	π_6			
5	π_7	π_7	π_7	π_7	π_7			
6			S_4	S_5			9	3
7	π_3	S_8	π_3	π_3	π_3			
8	π_5	π_5	π_5	π_5	π_5			
9	π_1		S_4	S_5	π_1			7

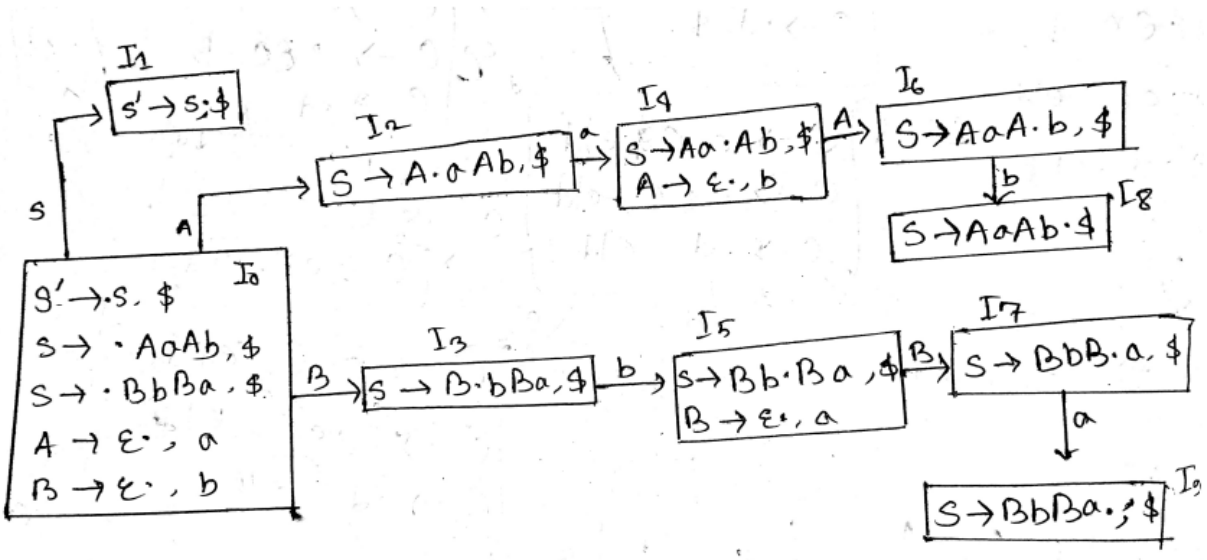
E → E + T → (1)
E → T (2)
T → TF (3)
T → F (4)
F → F * (5)
F → a (6)
F → b (7)

Construct CLR parsing table for the following grammar:

S → AaAb | BbBa

A → ε

B → ε



S → AaAb ⇒ π1

S → BbBa — π2

A → ε — π3

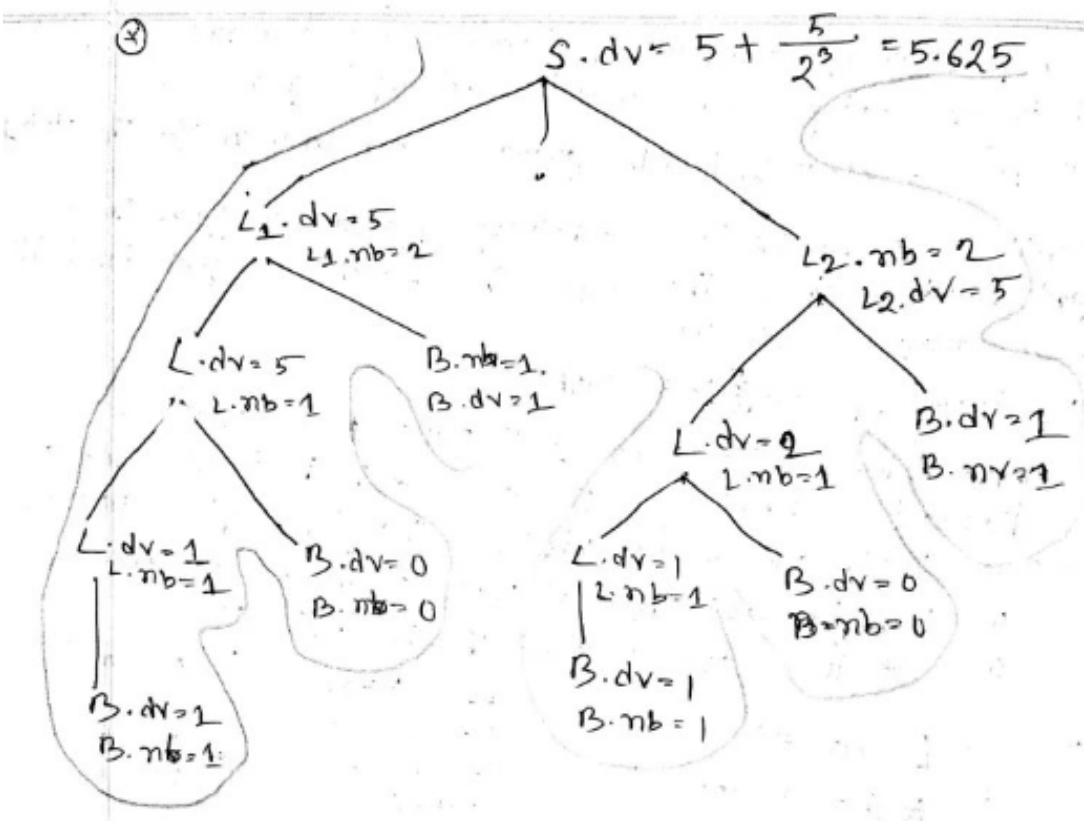
B → ε — π4

STATE	ACTION			GOTO		
	a	b	$	S	A	B
0	π3	π4		1	2	3
1			Accept			
2	S4					
3		S5				
4		π3			6	
5	π4					7
6		S8				
7	S9					
8			π1			
9			π2			

Show an annotate parse tree for the input expression **101.101** according to the following syntax-directed definition that converts binary to decimal with fraction.

Production	Semantic Rule
$S \rightarrow L_1.L_2$	$\{S.dv = L_1.dv + \frac{L_2.dv}{2^{L_2.nb}}\}$
$L \rightarrow L.B$	{L.dv = 2*L.dv+B.dv L.nb = L.nb+B.nb}
$L \rightarrow B$	{L.dv = B.dv L.nb = B.nb}
$B \rightarrow 0$	{B.dv = 0 B.nb = 1}
$B \rightarrow 1$	{B.dv = 1 B.nb = 1}

$L \rightarrow LB$



- **Contrast quadruples and triples with an example.**
- **Define indirect triple with an example.**

Quadruples

It is a structure which consists of 4 fields namely operator, op1, op2 and result.

op = operand

Pros:

Triples

This representation doesn't make use of extra temporary variable to represent a single operation.

Pros:

- Statement movement possible
- Quickly access value of temporary variables

Cons:

- Memory inefficient

-(a*b)+(c*d+e)

	Operator	Op1	Op2	Result
0	*	a	b	t_1
1	-	t_1		t_2
2	*	c	d	t_3
3	+	t_3	e	t_4
4	+	t_2	t_4	t_5

Indirect Triples

This representation makes use of pointer to the listing of all references to computations which is made separately and stored.

	Operator	Op1	Op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

100	(0)
101	(1)
102	(2)
103	(3)
104	(4)

Pros

- Statement movement possible

Cons

- Memory efficient compared to quadruples

Cons

- Statement movement is not possible

	Operator	Op1	Op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

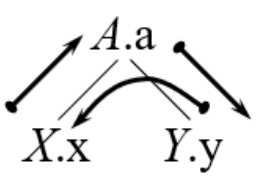
- Two memory reference is required

"Dependency graph should not contain any cycle"-why? Show an annotate parse tree for the input expression ***float a, b, c*** according to the following syntax-directed definition that stores type information into symbol table:

Production	Semantic Rule
$D \rightarrow D_1$, **id**	*{Addtype(id, D_1.type)* *D.type=D_1.type}*
$D \rightarrow T$ **id**	*{Addtype(id, T.type)* *D.type=T.type}*
$T \rightarrow$ *int*	*T.type =int*
$T \rightarrow$ *char*	*T.type =char*
$T \rightarrow$ float	*T.type =float*

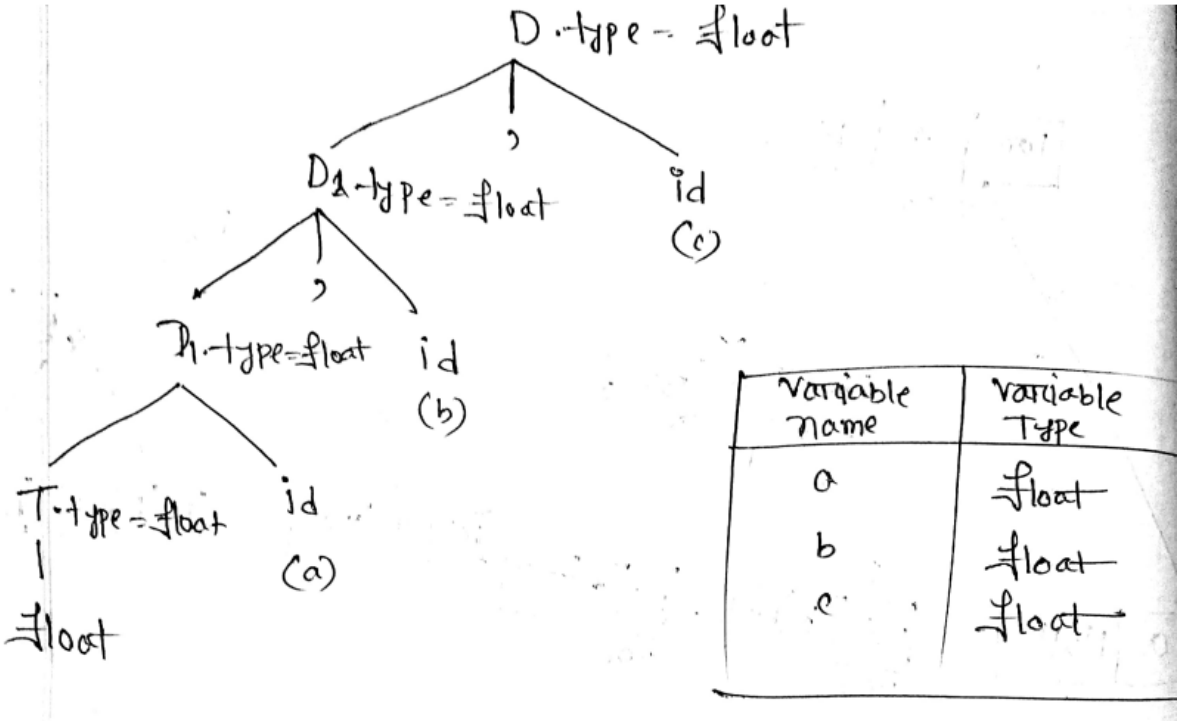
A dependency graph is used to represent the flow of information among the attributes in a parse tree.

A dependency graph cannot be cyclic **because cycles, or circular dependencies, make it impossible to evaluate the objects in the graph in a valid order**.



$$A.\mathrm{a} := f(X.\mathrm{x})$$
$$X.\mathrm{x} := f(Y.\mathrm{y})$$
$$Y.\mathrm{y} := f(A.\mathrm{a})$$

Error: cyclic dependence



Variable name	Variable Type
a	float
b	float
c	float

What is backpatching? Write down three-address code for the following segment of C code:

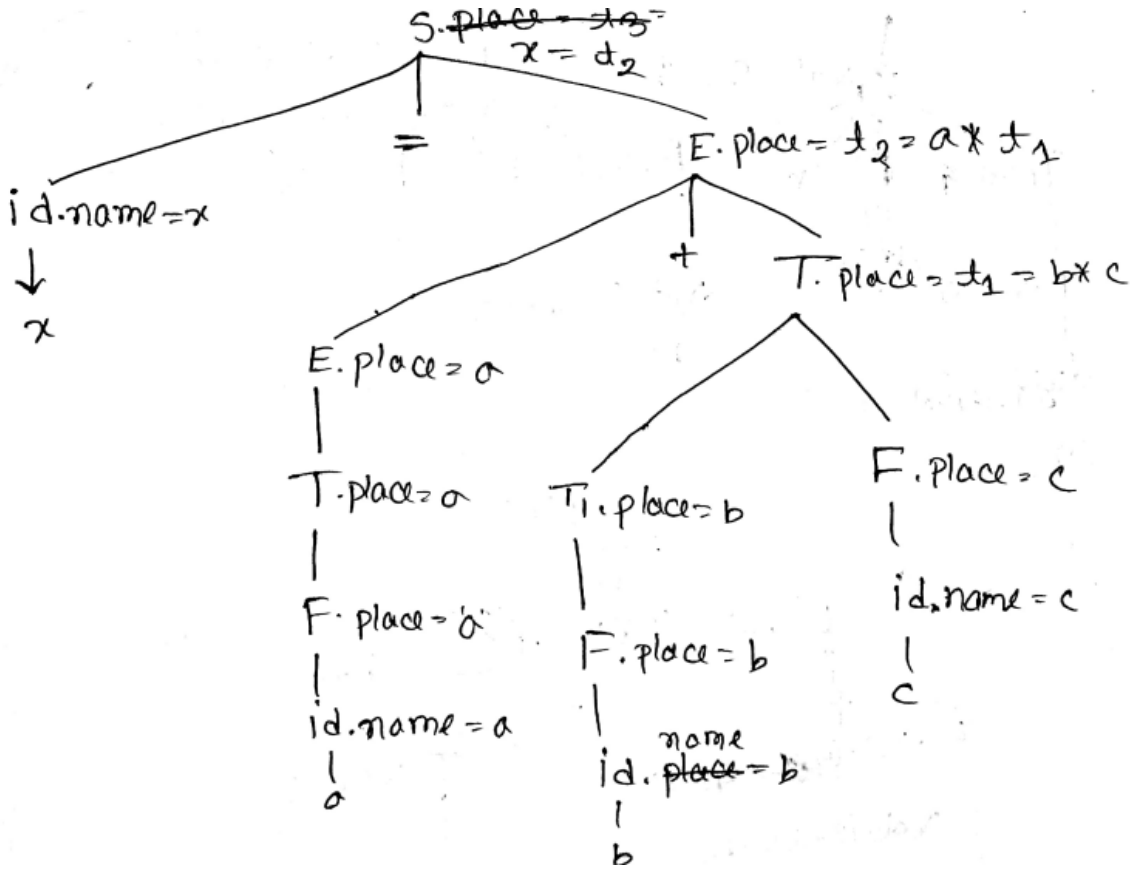
```
c = 0
do
{
if (a < b) then
x++;
else
x–;
c++;
} while (c < 5)
```

Backpatching is basically a pr**ocess of fulfilling unspecified information**.
Backpatching is a method to deal with jumps in the control flow constructs like if statements, loop etc. in the intermediate code generation phase of the compiler.

1. $c = 0$
2. $if\ (a < b)\ goto\ 4$
3. $goto\ 7$
4. $T1 = x + 1$
5. $x = T1$
6. $goto\ 9$
7. $T2 = x - 1$
8. $x = T2$
9. $T3 = c + 1$
10. $c = T3$
11. $if\ (c < 5)\ goto\ 2$
12.

Show an annotate parse tree for the input expression $x = a + b * c$ according to the following syntax-directed definition that generates three address code:

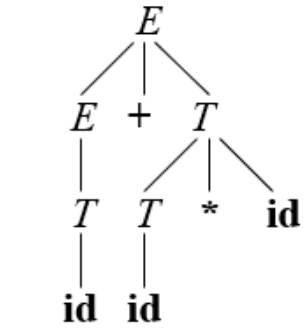
Production	Semantic Rule
$S \rightarrow id=E$	{gen(id.name, E.place);}
$E \rightarrow E+T$	{E.place=newtemp(); gen(E.place=E.place+T.place);}
$E \rightarrow T$	{ E.place=T.place}
$T \rightarrow T_1*F$	{T.place=newtemp(); gen(T.place=T_1.place*F.place);}
$T \rightarrow F$	{ T.place=F.place}
$F \rightarrow$ id	{ F.place=id.name}



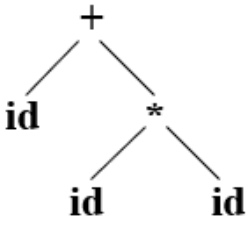
- **What do you mean by concrete and abstract syntax tree? Explain with examples.**

A parse tree is called a concrete syntax tree. A parse tree pictorially shows how the start symbol of a grammar derives a string in the language.

An abstract syntax tree (AST) is defined by the compiler writer as a more convenient intermediate representation. AST only contains semantics of the code.



Concrete syntax tree

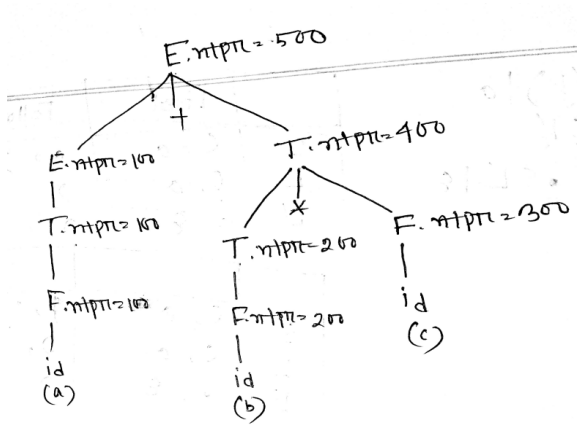


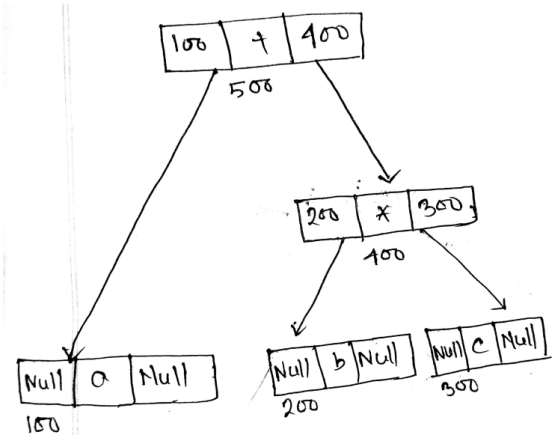
Abstract syntax tree

Show an annotate parse tree for the input expression $a + b * c$ according to the following syntax-directed definition that generates three address code:

Syntax tree

Production	Semantic Rule
$E \rightarrow E+T$	{E.nptr=mknode(E.nptr,+,T.nptr)}
$E \rightarrow T$	{E.nptr=T.nptr}
$T \rightarrow T*F$	{T.nptr=mknode(T.nptr,*,F.nptr)}
$T \rightarrow F$	{T.nptr=F.nptr}
$F \rightarrow id$	{F.nptr=mknode(Null,id,Null)}





What is a transition diagram? Explain with example. 3

2. a) Define the following terms: 2+2
   (i) Synthesized translation
   (ii) Inherited translation

   b) Give the parse tree and translations for the expression (4*7 + 19) * 2 according to the following syntax-directed translation scheme, in which *E.VAL* is an integer-valued translation. 4

Production	*Semantic Action*
$E \rightarrow E^{(1)} + E^{(2)}$	$\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$
$E \rightarrow digit$	$\{E.VAL := digit\}$

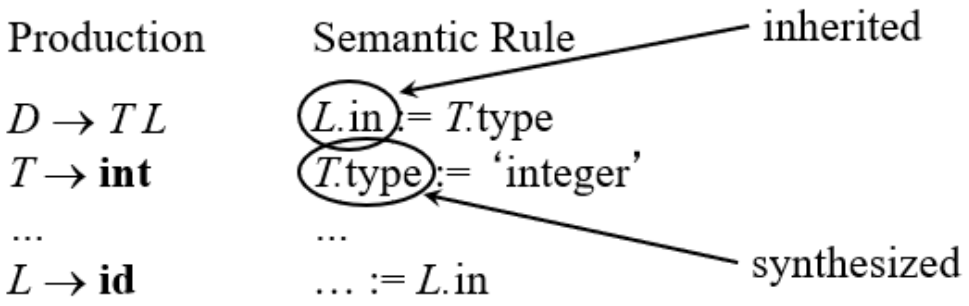
   Here *digit* stands for any digit between 0 and 9.

   c) What is intermediate code? What types of intermediate code are often used in compilers? 2

(a)

Synthesized: An attribute is said to be synthesized attribute if its parse tree node value is determined by the attribute at child nodes.

Inherited: An attributed is said to be inherited attribute if its parse tree node is determined by the attribute value at parent and/or siblings node.



(b) ...

(c) Intermediate Code is a form that serves as a connection between the front end and back end of a compiler, representing the program during various phases Intermediate code can translate the source program into the machine program.

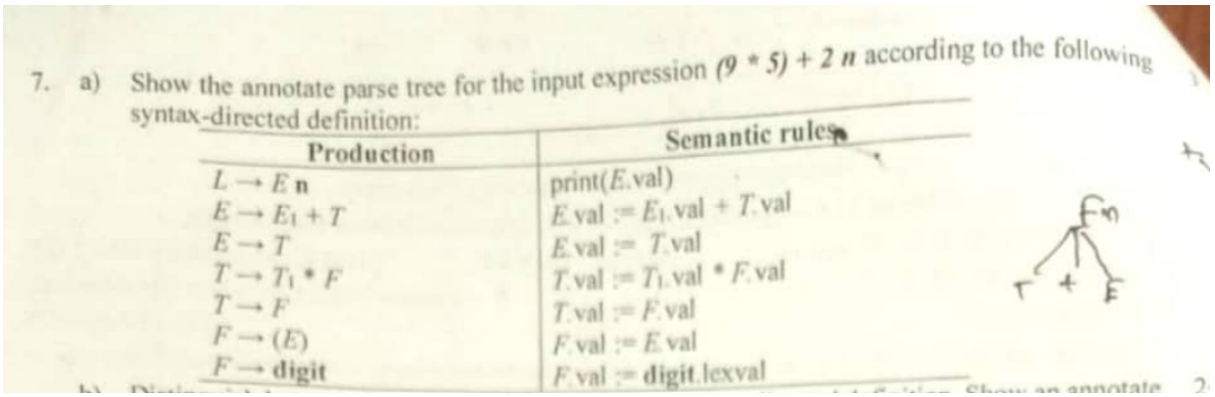
Types of intermediate code:

Linear form

- Postfix notation
- Three address code: A three address statement involves a maximum of three references, consisting of two for operands and one for the result.

Tree

- Syntax tree/Abstract Syntax Tree
- Directed Acyclic Graph



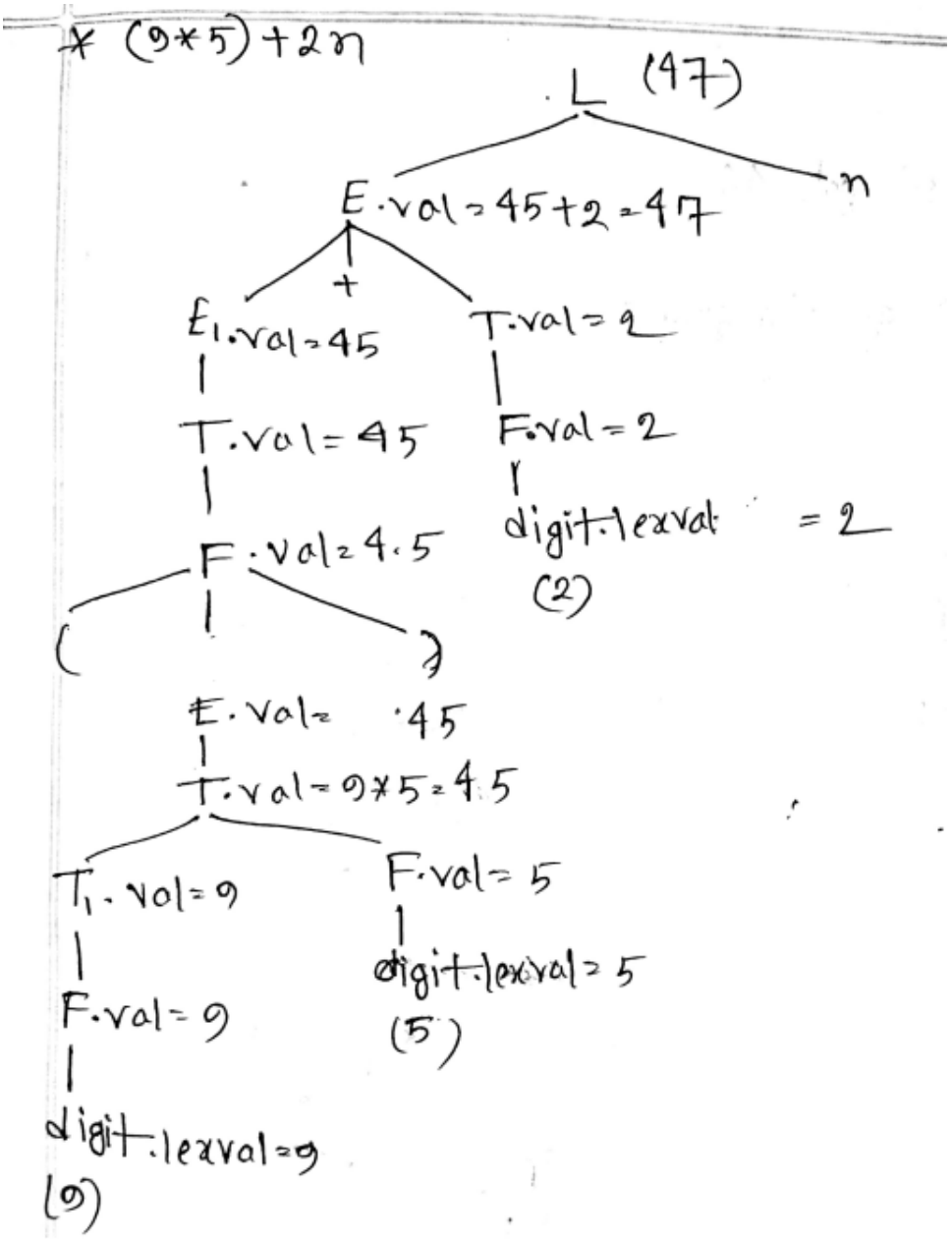
7. a) Show the annotate parse tree for the input expression *(9 * 5) + 2 n* according to the following syntax-directed definition:

Production	Semantic rules
$L \rightarrow E$ **n**	print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ **digit**	F.val := **digit**.lexval

7. a) Construct an annotate parse tree for the input expression (4 * 7 + 1) * 2 according to the following syntax-directed definition:

Production	Semantic rules
$L \rightarrow E$ **n**	print(E.val)
$E \rightarrow E_1 + T$	E.val := E_1.val + T.val
$E \rightarrow T$	E.val := T.val
$T \rightarrow T_1 * F$	T.val := T_1.val * F.val
$T \rightarrow F$	T.val := F.val
$F \rightarrow (E)$	F.val := E.val
$F \rightarrow$ **digit**	F.val := **digit**.lexval

b) What are the differences between



8. a) Show the expression – $(a * b) + (c * d + e)$ into quadruples and triples. Outline the advantages and disadvantages of quadruples and triples.

b) Evaluate three address code for the following code using backpatching:

```
sum=0;
for(i=1; i<11; i++){
    for(j=0; j<i; j++){
        x=A[i][j];
        sum=sum+x;
    }
}
```

c) Define DAG. Show a DAG for the following three address code:

$d = b * c$
$e = a + b$
$b = b * c$
$a = e - d$

(a)

	Operator	Op1	Op2	Result
0	*	a	b	t_1
1	-	t_1		t_2
2	*	c	d	t_3
3	+	t_3	e	t_4
4	+	t_2	t_4	t_5

Quadruples

Pros:

- Statement movement possible
- Quickly access value of temporary variables

Cons:

- Memory inefficient

	Operator	Op1	Op2
0	*	a	b
1	-	(0)	
2	*	c	d
3	+	(2)	e
4	+	(1)	(3)

Triples

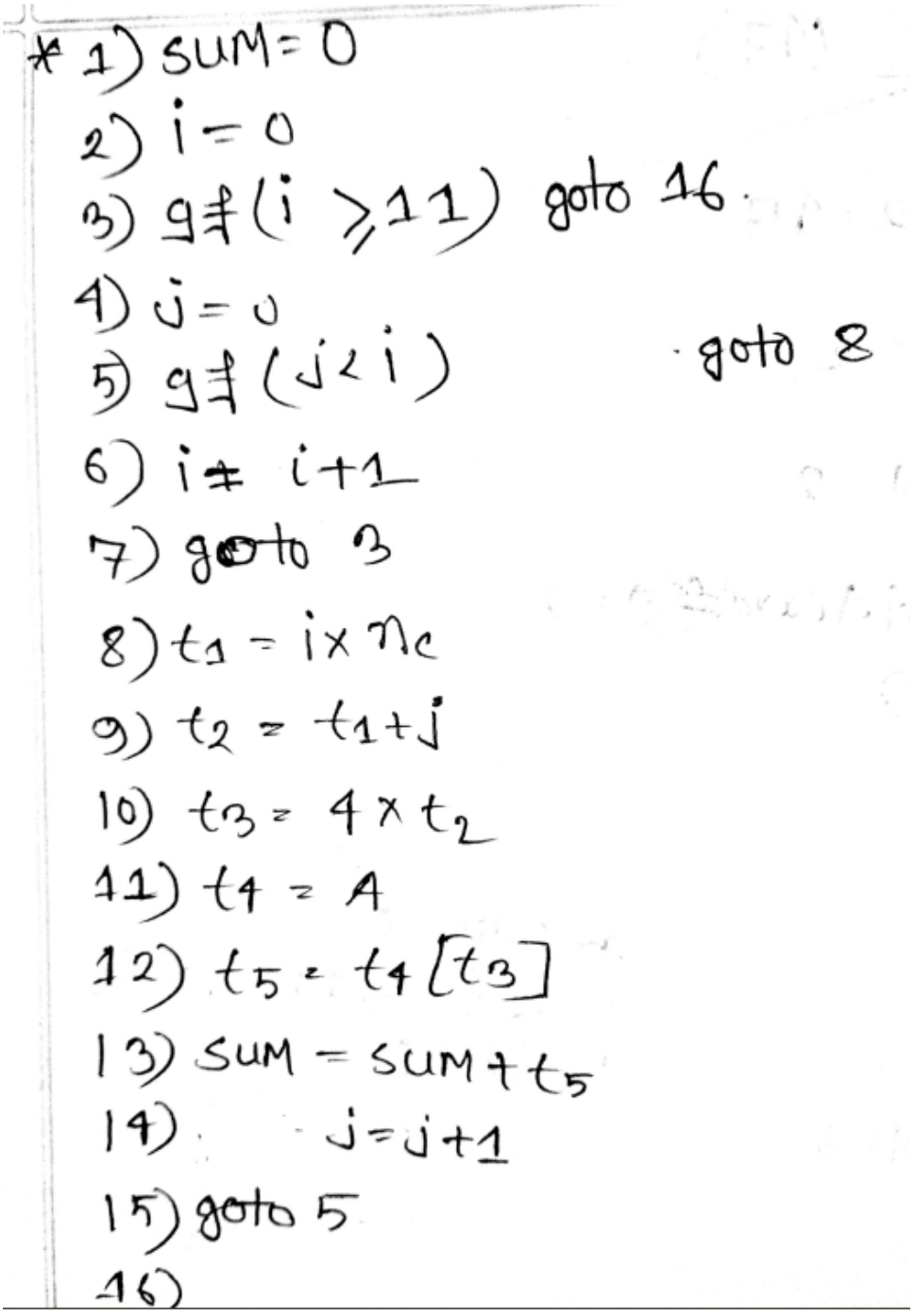
Pros:

- Memory efficient compared to quadruples

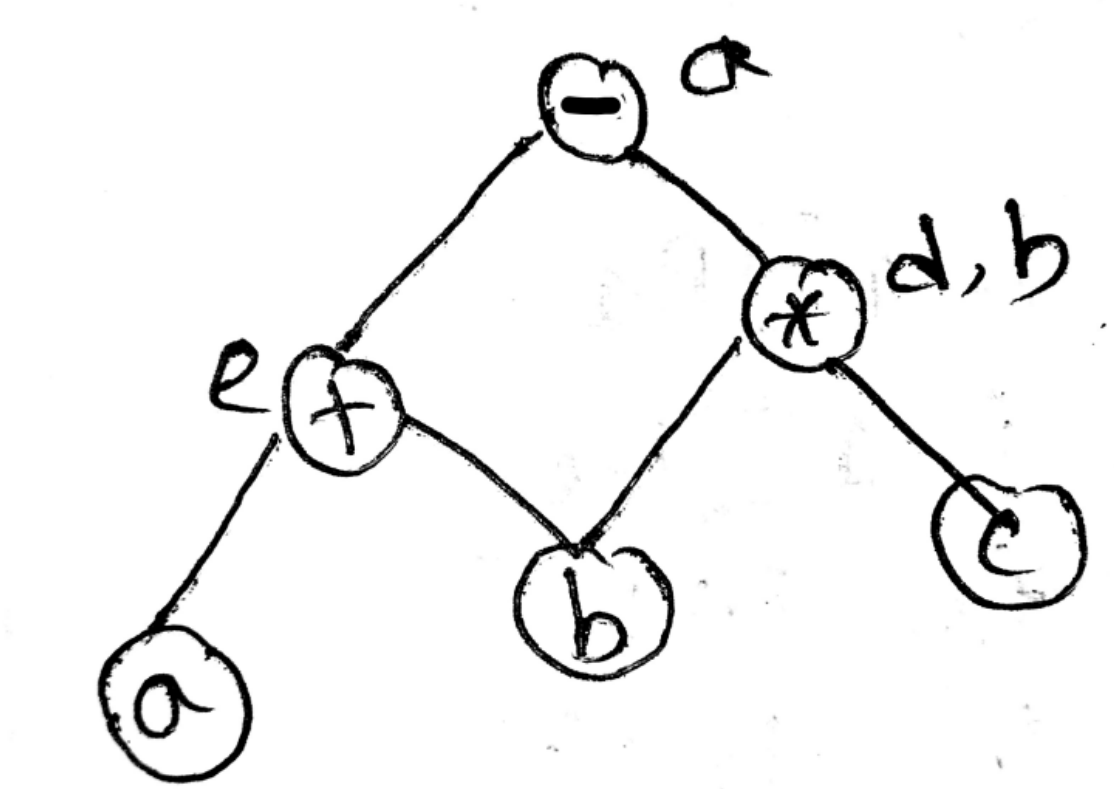
Cons

- Statement movement is not possible

(b)


* 1) sum = 0
2) i = 0
3) if (i >= 11) goto 16
4) j = 0
5) if (j < i) goto 8
6) i = i + 1
7) goto 3
8) t1 = i x n
9) t2 = t1 + j
10) t3 = 4 x t2
11) t4 = A
12) t5 = t4 [t3]
13) sum = sum + t5
14) j = j + 1
15) goto 5
16)

(c)



9. a) What are the types of code optimization techniques? Illustrate how the loop optimization performed in the code generation phase of a compiler. 1+2

   b) Write down how to find leaders in a basic block. Consider the following three address code and then design program flow graph (PFG): 1.5+2.5

```
1.  i = 1
2.  j = 1
3.  t1 = 5 * i
4.  t2 = t1 + j
5.  t3 = 4 * t2
6.  t4=t3
7.  a[t4]= -1
8.  j = j + 1
9.  if (j <= 5) goto 3
10. i = i + 1
11. if (i < 5) goto 2
12.
```

   c) Discuss about liveness analysis. Outline when you can say that X is a live variable (Assume X is a variable at a statement S_i). 3

(a)

Types of Code Optimization techniques

1. **Machine Independent:** This code optimization phase attempts to improve the intermediate code to get better target code as the output.
   a. Loop Optimization

b. **Constant Folding:** Replacing an expression that can be computed at compile time by its value. Example: x=10+5 → x=15

c. Redundancy Elimination

d. **Strength Reduction :** Replacing an expensive operator by cheaper one. Example: x/2 → x*0.5, A*2 → A << 1, x^2 → x*x

e. Algebraic simplification: *Example :* x+0=0+x=x, x-0=x

2. **Machine Dependent:** It is done after target code has been generated and when the code is transformed according to the target machine architecture.

   a. Register Allocation

   b. Use of addressing modes

   c. Peephole optimization

Loop Optimization: Loop optimization in code generation involves applying techniques to make loops run more efficiently.

- **Frequency Reduction/Code Motion:** A statement or expression which can be moved outside the loop body without affecting the semantic of the program.

```
// Original
for (int i = 0; i < n; i++) {
    int temp = a + b;
    array[i] = temp * i;
}
```

```
// Optimized
int temp = a + b;  // Moved o
for (int i = 0; i < n; i++) {
    array[i] = temp * i;
}
```

- **Loop Unrolling:** Reducing the number of times comparison are made in the loop.

```
for(i=0;i<10;i++){
    printf("Hi");
}
```

```
for(i=0;i<10;i=i+2){
    printf("Hi");
    printf("Hi");
}
for(i=0;i<10;i=i+2){
    printf("Hi");
     printf("Hi");
}
```

- **Loop jamming:** Combine or merge the bodies of two loops.

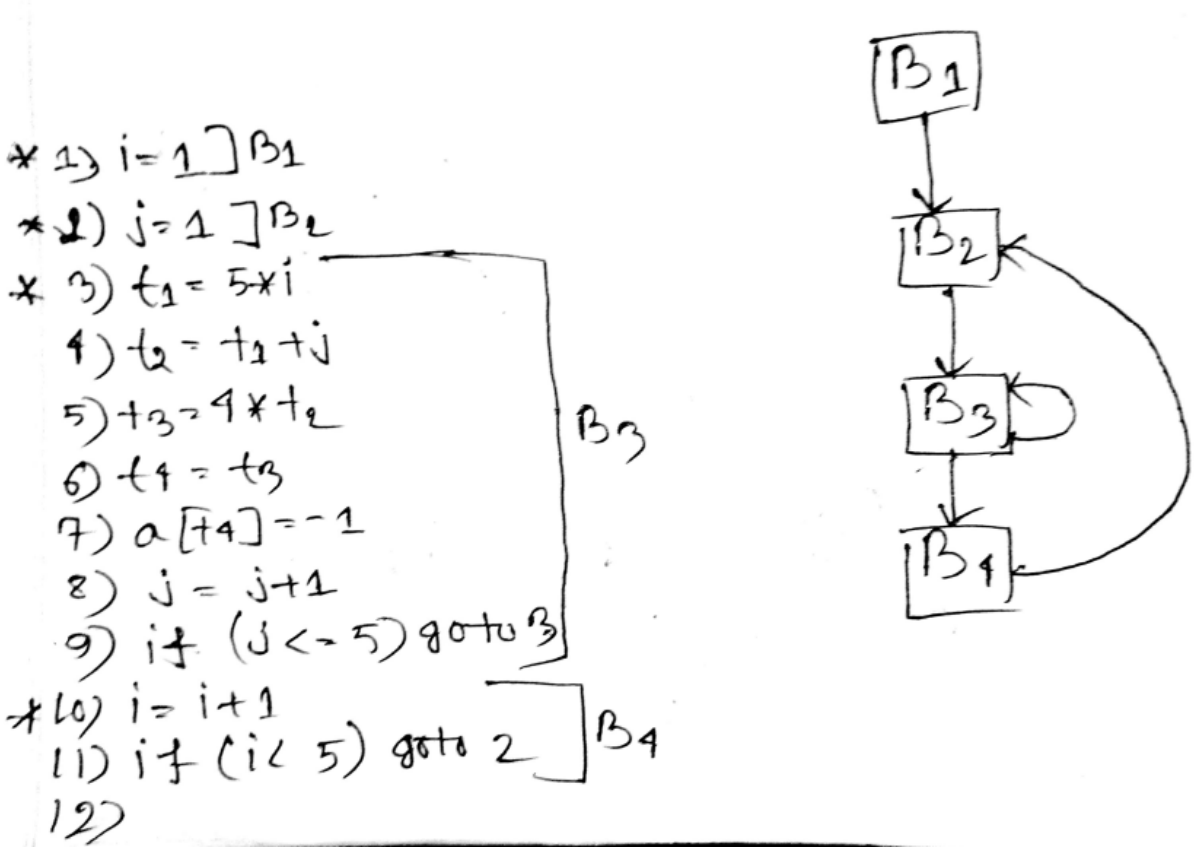
```
for(i=0;i<5;i++){
    a=i+5;
}
for(i=0;i<5;i++){
    b=i+10;
}
```

```
for(i=0;i<5;i++){
    a=i+5;
    b=i+10;
}
```

(b)

Finding leaders in a basic block

- The first three address instruction in the intermediate code is a leader
- Any instruction that is the target of conditional or unconditional jumps is a leader
- Any instruction that immediately follows a conditional and unconditional jumps is a leader



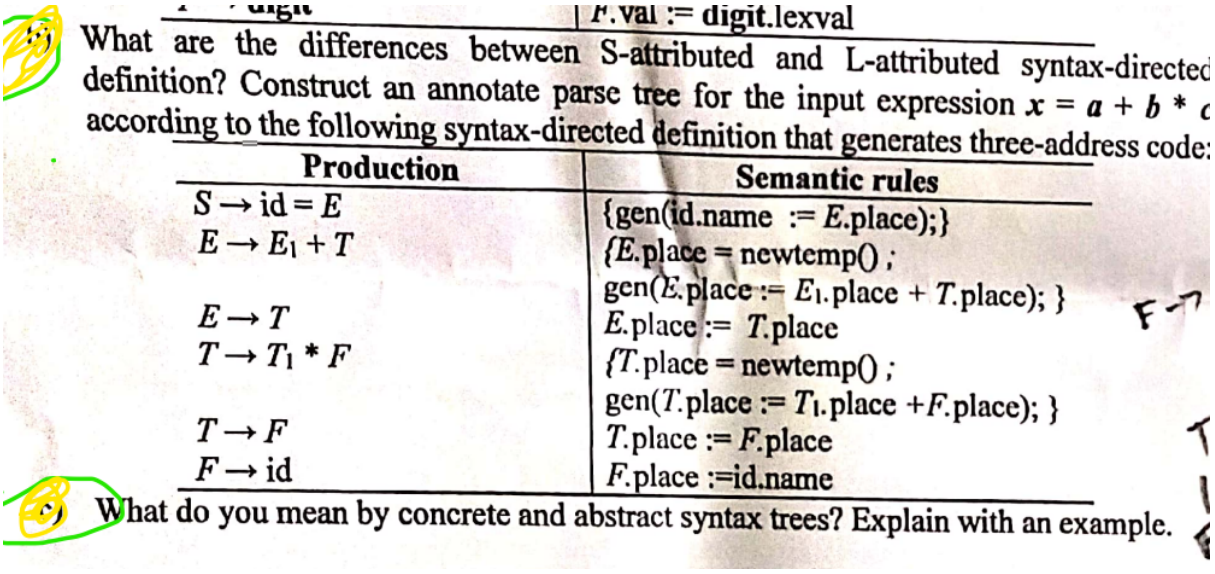
(c)

Liveness analysis or register allocation is a machine dependent optimization technique. The purpose of it is assigning multiple variable to a single register without

changing the program behavior.

X is a live variable at statement S_i iff

1. There is a statement of S_j using X
2. There is a path from S_i to S_j
3. There is no new definition to X before S_j



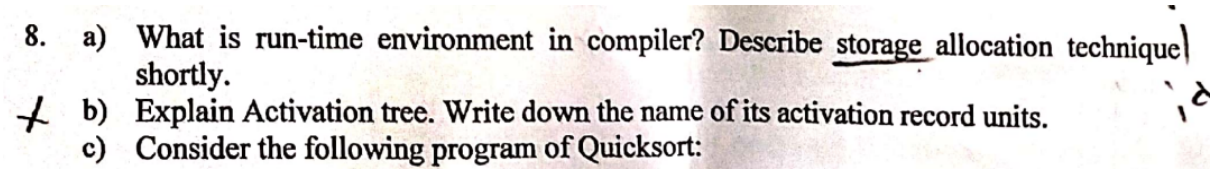
digit | F.val := digit.lexval

What are the differences between S-attributed and L-attributed syntax-directed definition? Construct an annotate parse tree for the input expression $x = a + b * c$ according to the following syntax-directed definition that generates three-address code:

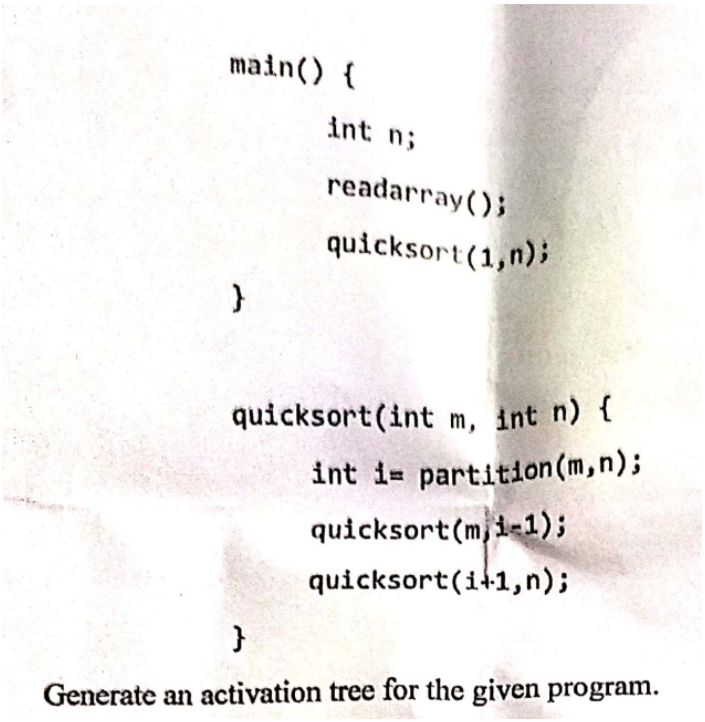
Production	Semantic rules
$S \rightarrow \text{id} = E$	{gen(id.name := E.place);}
$E \rightarrow E_1 + T$	{E.place = newtemp() ; gen(E.place := E_1.place + T.place); }
$E \rightarrow T$	E.place := T.place
$T \rightarrow T_1 * F$	{T.place = newtemp() ; gen(T.place := T_1.place +F.place); }
$T \rightarrow F$	T.place := F.place
$F \rightarrow \text{id}$	F.place :=id.name

What do you mean by concrete and abstract syntax trees? Explain with an example.

S-Attributed SDD	L-Attributed SDD
A SDD that uses only synthesized attribute is called S-attributed SDD. Ex: A → BCD {A.i=B.i; A.i=C.i; A.i=D.i}	A SDD that uses both synthesized and inherited attributes is called L-attributed SDD but each inherited attribute is restricted to inherit from parent or left siblings only. Ex: A → BCD {C.i=A.i; C.i=B.i} Not C.i=D.i
Semantic actions are always placed at right end of the production.	Semantic actions are placed anywhere on the R.H.S of the production.
Attributes are evaluated with Bottom up parsing.	Attributes are evaluated by traversing parse tree using depth-first, left to right.



8. a) What is run-time environment in compiler? Describe storage allocation technique shortly.
   b) Explain Activation tree. Write down the name of its activation record units.
   c) Consider the following program of Quicksort:



```
main() {
        int n;
        readarray();
        quicksort(1,n);
}

quicksort(int m, int n) {
        int i= partition(m,n);
        quicksort(m,i-1);
        quicksort(i+1,n);
}
```

Generate an activation tree for the given program.

(a)

The run-time environment in a compiler is the **setup** that created by the compiler to manage program execution. It includes the structures and mechanisms that support that function calls, variable storage, dynamic memory management and overall resource management.

Allocation Technique	Description	Use Case	Advantages	Limitations
Static Allocation	- Fixed memory at compile-time	Global and static variables	No run-time overhead	- No support for dynamic sizes - Doesn’t support dynamic data structure - recursion not supported
Stack Allocation	LIFO allocation for function calls	Local variables, function calls	- Efficient for function calls - Recursion supported	- Limited to static sizes - Doesn’t support dynamic data structure
Heap Allocation	Dynamic memory at run-time	Dynamic data structures	- Flexible, supports dynamic sizes - Allocation and deallocation will	Requires careful management

Allocation Technique	Description	Use Case	Advantages	Limitations
			be done at anytime based on user requirement - Recursion supported	

(b)

An activation tree is a **conceptual tool** used in compiler design and program analysis **to represent the sequence of function** or procedure calls in a program, based on its control follow.

Properties

- Each node represents an activation of a procedure
- The root shows the activation of the main function
- The node for procedure x is the parent of node for procedure y if and only if the control flows from x to procedure y.

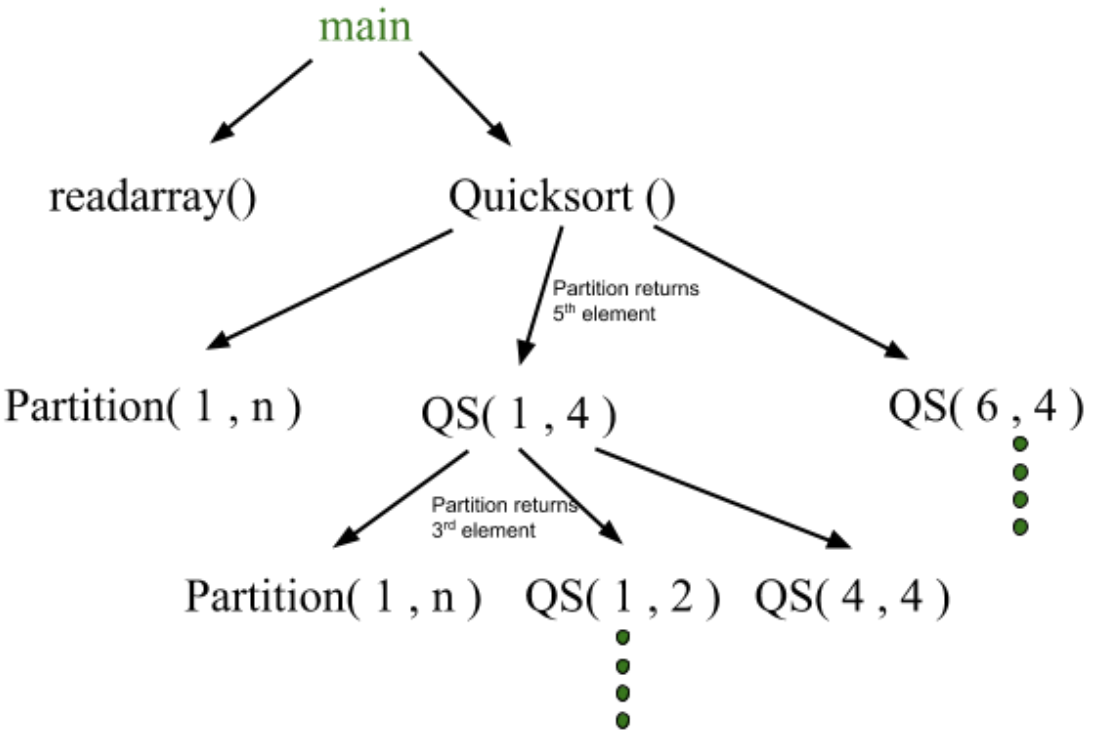
An activation record is the **contiguous block of storage** that **manages information** required by **a single execution** of a procedure.

Activation Record Units

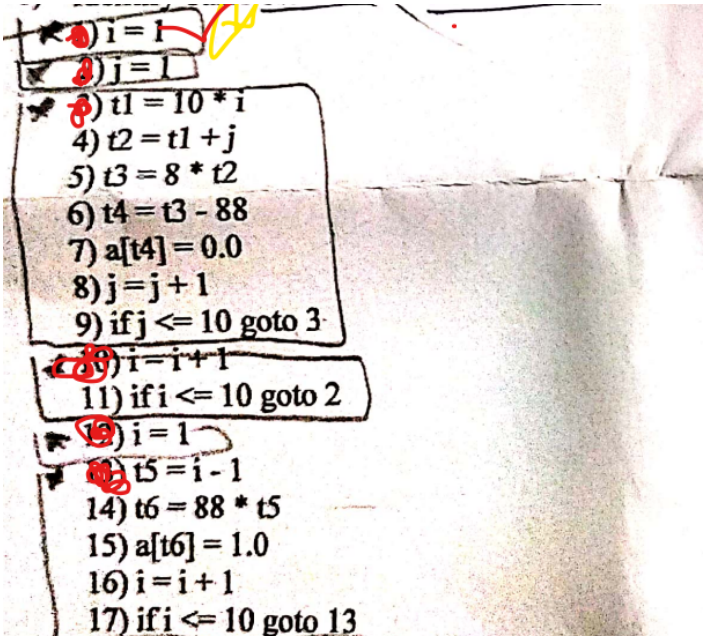
- **Temporaries:** The **temporary values**, such as those **arising in the evaluation** of expression, are stored in the field for temporaries
- **Local data:** The field for local data holds data that is **local to an execution of a procedure.**
- **Save Machine States:** The field for Saved Machine Status **holds information** about the state of the machine **just before the procedure** is called.
- **Access Link:** It refers to information in other activation records that is not local. The main purpose of this is to access the data which is not present in the local scope of the activation record. (In which outer function the function is defined)
- **Control Link:** It refers to an **activation record of the caller**. They are used for links and saved status. (Which function is called in the function)
- **Parameter List:** The field fo**r parameters list is used by the calling procedure parameters** to supply parameters to the called procedure.

- **Return value:** The field for the return value is used by the called procedure **to return value to the calling procedure**.

(c)



9. a) What is peephole optimization? How is the optimization performed in the code generation phase of a compiler?
b) Identify basic blocks and draw flow graph for the following three-address code:



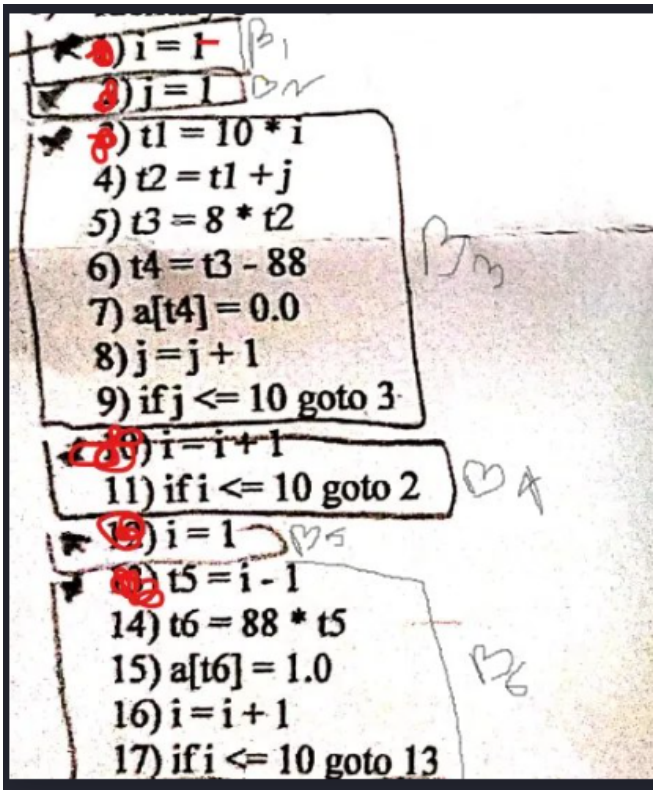
(a)

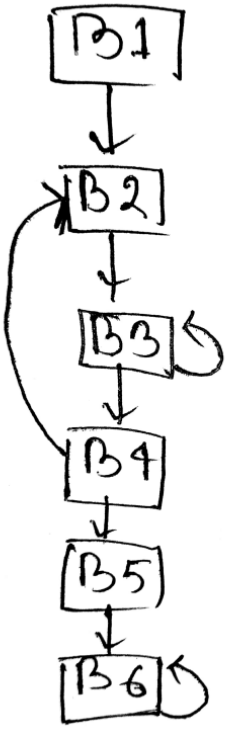
Peephole optimization is an optimization technique performed on a **small set of compiler-generated instructions**, known as a peephole or window.

How the optimization performed

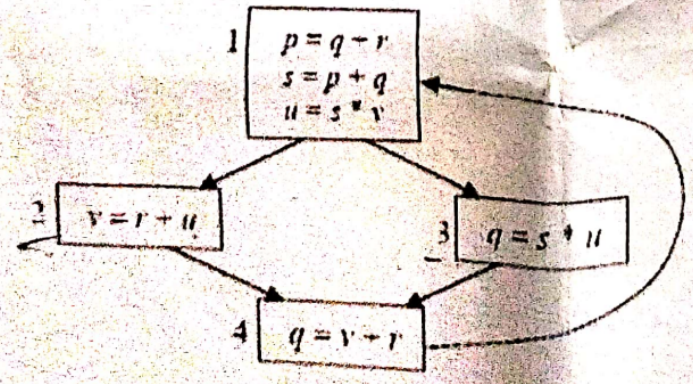
1. **Identify the Peephole:** Compiler finds the small sections of the generated code that needs optimization.
2. **Apply the Optimization rule:** After identification, the compiler applies a predefined set of optimization rules to the instruction in the peephole.
3. **Evaluate the result:** After applying the optimization rule, the compiler evaluates the optimized code to check whether the changes make the code better than the original in terms of speed, size or memory.
4. **Repeat:** The process is repeated by finding new peepholes and applying the optimization rule.

(b)

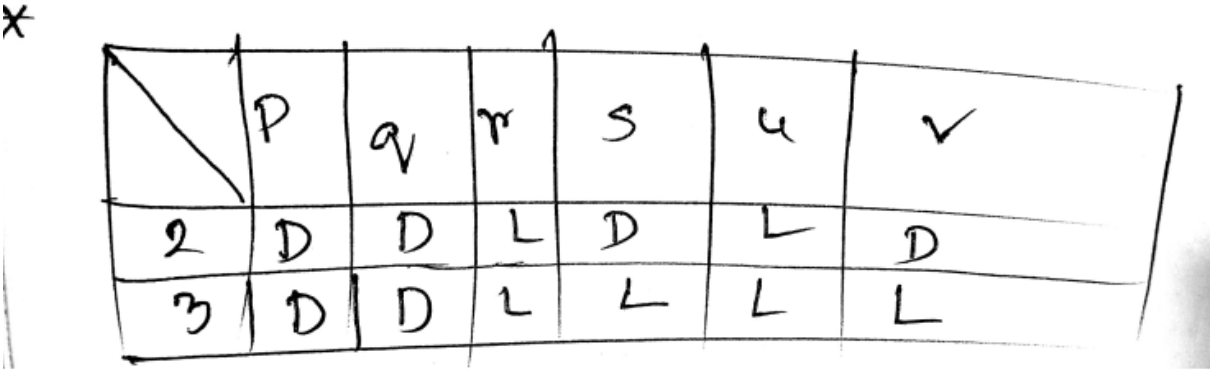




c) Consider the following flow graph:

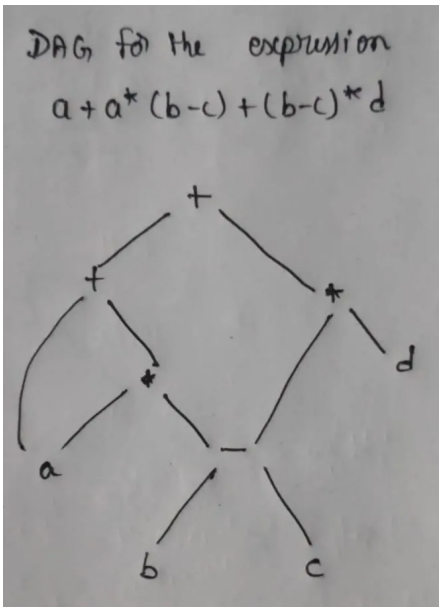


Show the live and dead variables of the basic blocks 2 and 3 using liveness analysis.



	p	q	r	s	u	v
2	D	D	L	D	L	D
3	D	D	L	L	L	L

- Show Syntax tree and DAG for the expression a + a * (b – c) + (b – c) * d. Show the benefits of DAG over syntax tree using three address code.



Benefits of using Directed Acyclic Graphs (DAGs) over syntax trees for generating Three Address Code, in point form:

1. **Redundancy Elimination**: DAGs remove duplicate sub-expressions by sharing common nodes, while syntax trees may repeat identical expressions.
2. **Memory Efficiency**: DAGs require less memory as they represent shared sub-expressions with a single node, unlike syntax trees, which store duplicates.
3. **Simplified Optimization**: DAGs make optimizations (like constant folding and common sub-expression elimination) easier, as shared computations are directly represented.

4. **Reduced Code Size**: Code generation from a DAG can produce fewer instructions since it avoids recomputing repeated expressions, whereas syntax trees often generate redundant code.
5. **Clear Evaluation Order**: DAGs clarify dependency relationships, making it easier to determine the correct order for computation, which is less straightforward with syntax trees.
6. **Improved Intermediate Representation**: DAGs focus on computational dependencies rather than structural syntax, streamlining transformations for compiler optimizations.