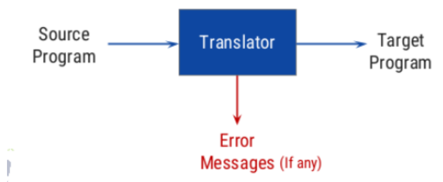
CHAPTER:1:

Language processor: A language processor is a software program designed or used to perform tasks such as processing program code to machine code.

Translator: A translator is a program that takes one form of program as input and converts it into another form.

Types of translators are:

1. Compiler
2. Interpreter
3. Assembler



Compiler: A compiler is a program that reads a program written in source language and translates it into an equivalent program in target language.

OR Compilers are basically use to convert high level language into low level language , here high level language means source code that we write in c,c++,java etc. and low level language means assembly language or machine code.

Interpreter: Interpreter is also a program that reads a program written in source language and translates it into an equivalent program in target language line by line.

Assembler: Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.

The Structure if a Compiler / Phases of a COMPILER: https://youtu.be/wnJMYuKaTvk

Language Processing System: https://youtu.be/Pwps_CP5MFE

Application of Compiler Technology:

- Compilation of Code: Compilers are the tools that convert source code into machine-specific code.
- High-level programming language implementation: A developer uses the language to specify an algorithm. The compiler then converts the program to the target language. Higher-level programming languages are simpler to write programs but are also less effective so to effectively implement high level languages we need compilers.
- Gaming: Compiler technology is widely used in game development. Compilers convert the game actions into machine code and thus provide better usability. Compilers also contribute to the security and stability of gaming software.
- Designing new computer systems: Compilers were created after the machines were built in the early days of computer design. Hence, when developing modern computer design, compilers are created during the processor-design phase.
- Program Translations: Compilers used in Program translation like Binary translation etc.
- Helps to make the code independent of the platform.
- Makes the code free of syntax and semantic errors.

Compiler vs Interpreter: https://youtu.be/fhgTETR77PY

The Role of Lexical Analyzer / Functions of a Lexical Analyzer: https://youtu.be/CD542K8DqCQ

Define Tokens, patterns and Lexeme *(ASKED IN UNIT TEST-1)* & Count the numbers of tokens: https://youtu.be/rRWzN219fsg

→Questions on Count the numbers of tokens: https://youtu.be/3NbakcFk2xg

Lexical Analyzer Generator LEX: https://youtu.be/R-RFzp9lfxM

QuestionBank Remaining Questions CH.1:

Explain the advantages of a compiler over an interpreter.

Execution Speed: Compiled programs generally tend to run faster than interpreted programs.

Larger Applications: For larger projects, compilers are often preferred because they can handle complex codebases more efficiently.

Error are shown instantly if we used compiler however in the interpreter it takes the time.

Compilers can generate object code significantly more effective and easily then the interpreter.

Overall, compilers are generally better suited for larger programs or projects where performance and efficiency are important.

Explain recognition of reserved words and identifiers.

- Reserved Words: Reserved words, also known as keywords, are words in a programming language that have special meanings and are used for specific purposes. These words are part of the language's syntax and are reserved for specific actions, statements, or declarations. For example, in the programming language Python, words like "if," "else," "while," and "for" are reserved words used for control flow and looping.

In compiler design, recognizing reserved words is crucial because they help the compiler understand the structure and meaning of the code being processed.

The role of the lexical analyzer (lexer) is to recognize these reserved words when they appear in the source code. Once recognized, the lexer generates a token for each reserved word encountered, associating it with its corresponding meaning in the language.
- Identifiers: Identifiers are names given by the programmer to variables, functions, classes, and other program elements. They provide a way to refer to different parts of the program. Identifiers are essential for writing readable and maintainable code.
The lexical analyzer's role is to recognize and extract identifiers from the source code. When an identifier is identified, it generates a token for it, storing the actual name of the identifier as a value along with the token type.
Define: alphabet, string and language and also define terms for parts of string.
Alphabet:An alphabet is a finite set of symbols or characters from which strings are formed. These symbols can be letters, numbers, special characters, or any other distinct elements.
String:A string is a sequence of symbols chosen from an alphabet. For example, in the English alphabet, "hello" is a string composed of the symbols 'h', 'e', 'l', 'l', and 'o'.
Language:In the context of formal languages, a language is a set of strings, where each string is formed from an alphabet. Languages can be finite or infinite.
Now, let's define some terms related to parts of a string:
Character:A character is a single symbol or element from the alphabet. For example, in the string "apple," each of the letters 'a', 'p', 'p', 'l', and 'e' are characters.
Substring:A substring is a contiguous sequence of characters taken from a larger string. For example, in the string "banana," "ana" is a substring.
Prefix:A prefix is a substring that appears at the beginning of a string. For instance, in the string "programming," both "pro" and "prog" are prefixes.
Suffix:A suffix is a substring that appears at the end of a string. In the string "algorithm," both "rithm" and "ithm" are suffixes.
Concatenation:Concatenation is the process of combining two or more strings to create a longer string. For instance, if you concatenate "hello" and "world," you get the string "helloworld."
Length: The length of a string is the count of characters it contains.
Empty String**:**An empty string is a string with no characters. It's often denoted by the symbol ε or simply "".
Palindrome**:**A palindrome is a string that reads the same forwards as it does backwards. For example, "level"
Define cross compiler, token and handle, yacc:
Cross Compiler: it's a compiler that runs on one type of computer system (the host) but produces code that can run on a different type of computer system (the target).
Token**:** Tokens represent the meaningful elements of the source code, such as keywords, identifiers, literals (numbers or strings), and operators. The lexical analyzer breaks down the input source code into tokens.
Handle: The handle in compiler design is referred to the substring of the right-hand side in a production rule in context-free grammar. When we are constructing a parse tree, the handle is the portion of the input string that has been matched. It can be reduced to a non-terminal symbol according to a production rule.
Yacc: Yacc (Yet Another Compiler Compiler) is a tool used for generating parsers, specifically LALR (Look-Ahead Left-to-Right, Rightmost Derivation) parsers, from formal grammar specifications. Yacc is often used in combination with lex (a lexical analyzer generator) to create the lexical and syntactic analysis phases of a compiler.
Write a short note on input buffering method. ***(ASKED IN UNIT TEST-1)***
There are mainly two techniques for input buffering:
Buffer pairs: The lexical analysis scans the input string from left to right one character at a time. So, specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.

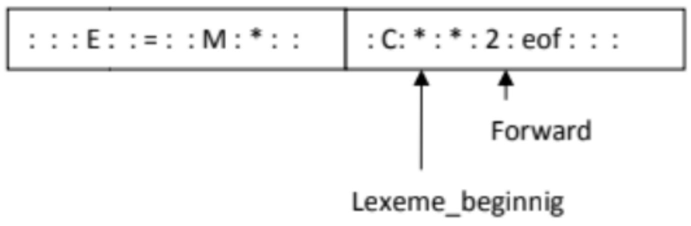


Fig. 2.2 An input buffer in two halves

- Buffer divided into two N-character halves, where N is the number of character on one disk block.
- There are two pointers: lexemeBegin and forward.
- Pointer Lexeme Begin, marks the beginning of the current lexeme.
- Pointer Forward, scans ahead until a pattern match is found.
- When a lexeme is discovered, lexeme begin is set to the character immediately after the newly discovered lexeme, and forward is set to the character at the right end of the lexeme.

Sentinels: In buffer pairs we must check, each time we move the forward pointer that we have not moved off one of the buffers. Thus, for each character read, we make two tests.
Test 1: Check for the buffer's end. Test 2: To figure out which character is being read.
By expanding each buffer in half to store a sentinel character at the end, sentinel reduces the two checks to one.
The sentinel is a special character that cannot be part of the source program, and a natural choice is the character EOF.

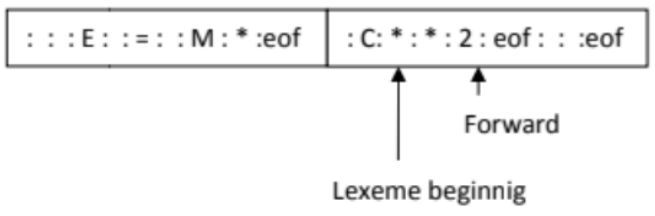
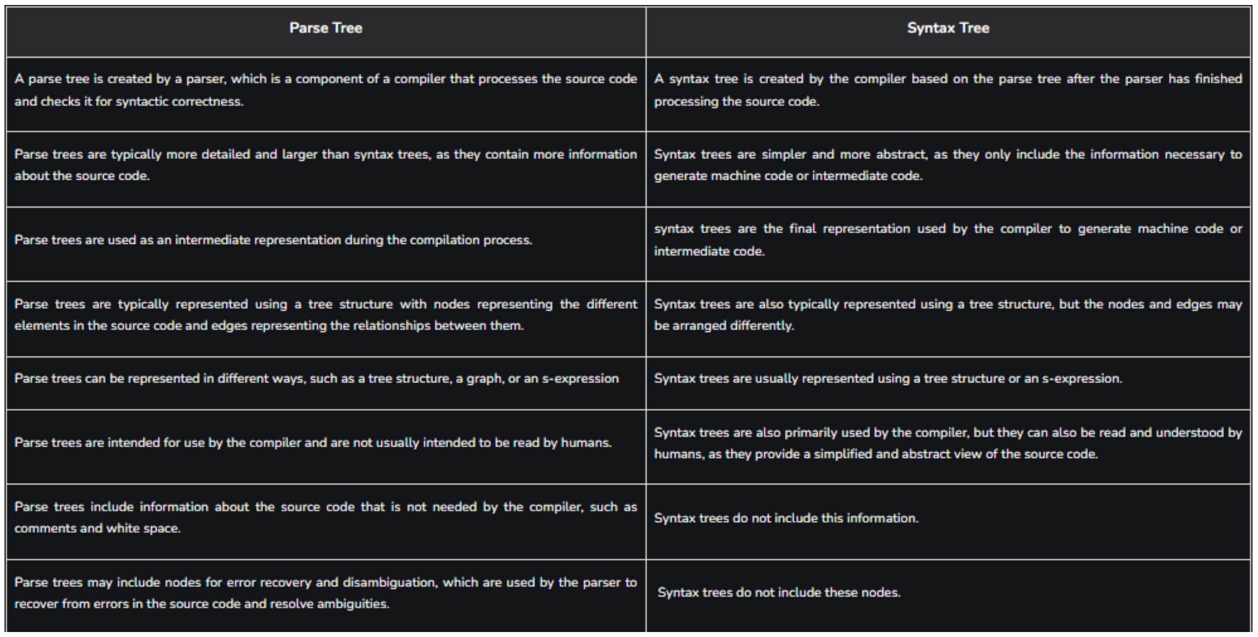


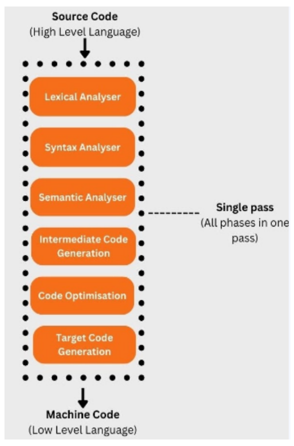
Fig.2.3. Sentinels at end of each buffer half

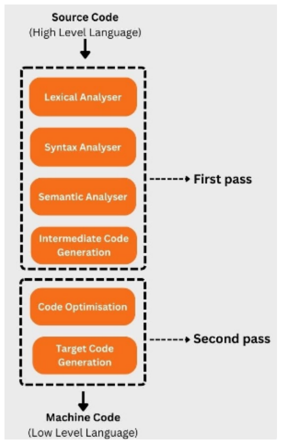
Difference between syntax tree and parse tree:



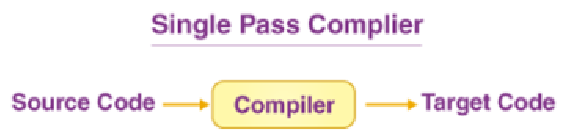
Parse Tree	Syntax Tree
A parse tree is created by a parser, which is a component of a compiler that processes the source code and checks it for syntactic correctness.	A syntax tree is created by the compiler based on the parse tree after the parser has finished processing the source code.
Parse trees are typically more detailed and larger than syntax trees, as they contain more information about the source code.	Syntax trees are simpler and more abstract, as they only include the information necessary to generate machine code or intermediate code.
Parse trees are used as an intermediate representation during the compilation process.	syntax trees are the final representation used by the compiler to generate machine code or intermediate code.
Parse trees are typically represented using a tree structure with nodes representing the different elements in the source code and edges representing the relationships between them.	Syntax trees are also typically represented using a tree structure, but the nodes and edges may be arranged differently.
Parse trees can be represented in different ways, such as a tree structure, a graph, or an s-expression	Syntax trees are usually represented using a tree structure or an s-expression.
Parse trees are intended for use by the compiler and are not usually intended to be read by humans.	Syntax trees are also primarily used by the compiler, but they can also be read and understood by humans, as they provide a simplified and abstract view of the source code.
Parse trees include information about the source code that is not needed by the compiler, such as comments and white space.	Syntax trees do not include this information.
Parse trees may include nodes for error recovery and disambiguation, which are used by the parser to recover from errors in the source code and resolve ambiguities.	Syntax trees do not include these nodes.

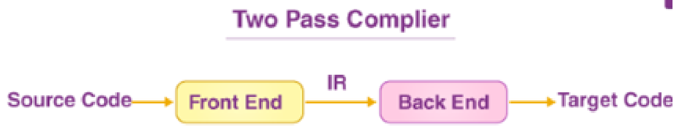
Explain the roles of Linker and Loader:
Linker**:** The main function of Linker is to generate executable files.The linker takes input of object code generated by compiler/assembler.
Loader**:** Whereas the main objective of Loader is to load executable files to main memory. And the loader takes input of executable files generated by linker.
Explain types of compiler.
One-pass compiler: It is a type of compiler that compiles whole process in one-pass.A Single-pass Compiler combines all the compiler phases in a single module.





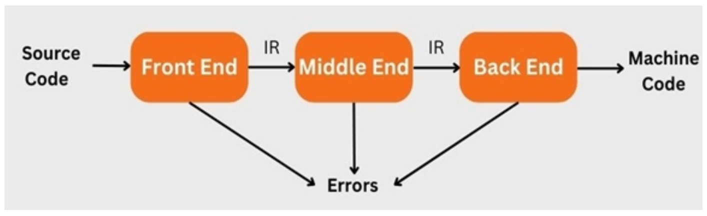
One-Pass Compiler *Two-Pass Compiler*





Two pass compiler: It is a type of compiler that compiles whole process in two-pass. It generates intermediate code.A Two pass compiler processes the source code into machine code in two passes.
- Front End: This is the analysis Phases of Compiler which involves scanning the source code, performing lexical analysis and finding the syntax errors. This phase generates an Intermediate Code which is passed to the Back end.
- Back End: Back end or the synthesis phase generates the machine code with symbol table representation and intermediate code.

Multi pass Compiler: A multi-pass compiler processes the source code in multiple passes. A huge program is broken into various small programs, and all of these programs run simultaneously. Numerous intermediate codes are generated, and the output of the last phase is an input of the current phase.



Incremental compiler: The compiler which compiles only the changed line from the source code and update the object code.
Cross compiler: It's a compiler that runs on one type of computer system (the host) but produces code that can run on a different type of computer system (the target).

CHAPTER:2:

Role of the Parser: https://youtu.be/CdC61wH1Zic
In our compiler model, the parser obtains a string of tokens from lexical analyzer,

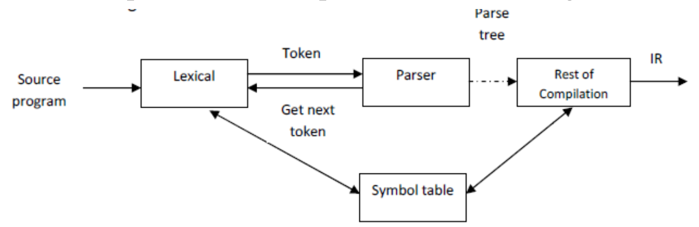
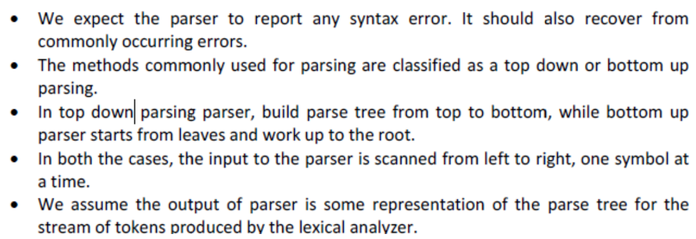


Fig.3.1.1. Position of parser in compiler model



- We expect the parser to report any syntax error. It should also recover from commonly occurring errors.
- The methods commonly used for parsing are classified as a top down or bottom up parsing.
- In top down parsing parser, build parse tree from top to bottom, while bottom up parser starts from leaves and work up to the root.
- In both the cases, the input to the parser is scanned from left to right, one symbol at a time.
- We assume the output of parser is some representation of the parse tree for the stream of tokens produced by the lexical analyzer.

Grammars: https://youtu.be/ETMcmKtAgBs
Types of Errors:
Lexical error: Lexical errors can be detected during lexical analysis phase.
Typical lexical phase errors are:
- Spelling errors in keywords
- Exceeding length of identifier or numeric constants
- Appearance of illegal characters

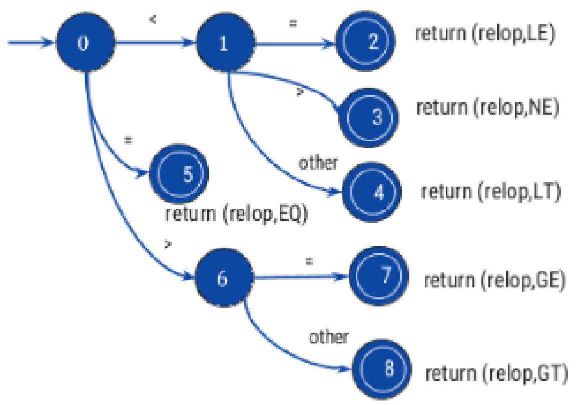
Syntax error: Syntax error appear during syntax analysis phase of compiler.
Typical syntax phase errors are:
- Errors in structure
- Missing operators
- Unbalanced parenthesis

Example: printf("Hello") so here semicolon is missing
Semantic error: Semantic error detected during semantic analysis phase.

Typical semantic phase errors are:
- Incompatible types of operands
- Undeclared variable

QuestionBank Remaining Questions CH.2:

Error recovery strategies (Ad-Hoc & systematic methods): There are mainly four error recovery strategies:
Panic mode: In this method on discovering error, the parser discards input symbol one at a time. This process is continued until one of a designated set (like end, semicolon) of synchronizing tokens (are typically the statement or expression terminators) is found.
Synchronizing tokens are delimiters such as semicolon or end.
These tokens indicate an end of the statement.
If there is less number of errors in the same statement then this strategy is best choice.
Phrase level recovery: In this method, on discovering an error parser performs local correction on remaining input. The local correction can be replacing comma by semicolon, deletion of semicolons or inserting missing semicolon.
This type of local correction is decided by compiler designer.
This method is used in many error-repairing compilers.
Error production: The use of the error production method can be incorporated if the user is aware of common mistakes that are encountered in grammar in conjunction with errors that produce erroneous constructs. If error production is used then, during parsing we can generate appropriate error message and parsing can be continued. Example: write 5x instead of 5*x
Global correction: In order to recover from incorrect input, the parser analyzes the whole program and tries to find the closest match for it, which is error-free. The closest match is one that does not do many insertions, deletions, and changes of tokens. This method is not practical due to its high time and space complexity.
Draw transition diagram for relational operators.: https://youtu.be/LQNBbajHxfo?si=evXFadpb-cg1F0oP



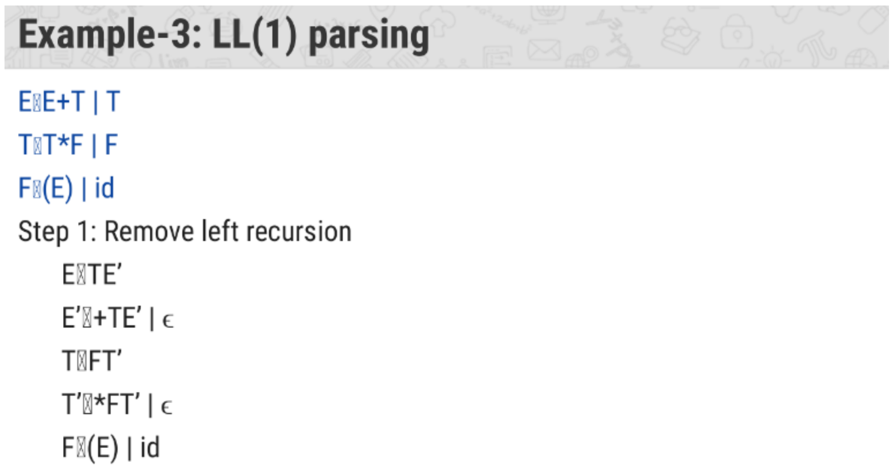
What is symbol table? For what purpose compiler uses symbol table?
Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
It is used by the compiler to achieve compile-time efficiency.
The symbol table helps the compiler resolve issues related to variable scoping.
During the early stages of compilation, the compiler builds the symbol table by scanning the source code.
The symbol table is used in detecting errors, such as undeclared variables or duplicate declarations. If a variable is referenced but not found in the symbol table, the compiler can issue an error message.
Explain role of finite automata in the compiler.
Finite automata play a crucial role in the lexical analysis phase of a compiler. Specifically, they are used to implement regular expressions and to recognize tokens in the source code
Finite automata are used to recognize and extract tokens from the source code in lexical analysis phase.
Lexer (Lexical Analyzer) uses finite automata to implement the regular expressions for various tokens.
Finite automata can also be used to identify and report lexical errors, This assists in early error detection.

CHAPTER:3:

Top down Parsing: https://youtu.be/1rGLFs5pGNM
https://youtu.be/mP6YNYSpZV4
BruteForce in Top Down Parsing / Recursive-Descent Parsing : https://youtu.be/4gKY5GHOMQE

First and follow: https://youtu.be/oCyHux4Ta3k
https://youtu.be/sWiXyhCwnsE
LL(1) grammar: https://youtu.be/BNfNBYnKjp8
***NOTE:* Remove Left Recursion**
In Top Down parser means in LL1 if in production rule if Left Recursion is there then we need to convert it into right recursion then we can proceed ahead because A top-down parser cannot handle left recursive productions.
So Convert Left Recursion into Right Recursion: https://youtu.be/nJxF9M2lFEw
https://youtu.be/ttplIT5r8OQ
https://youtu.be/T96Jua67drw

QUESTION ON LL1:
Here in Question Left Recursion is there so first remove left recursion then make LL1 table.



Example-3: LL(1) parsing

E→E+T | T
T→T*F | F
F→(E) | id
Step 1: Remove left recursion
E→TE’
E’→+TE’ | ϵ
T→FT’
T’→*FT’ | ϵ
F→(E) | id

Now Question becomes same as this : https://youtu.be/BNfNBYnKjp8

LL(1) Grammar Identification Short Cut Trick: https://youtu.be/StdedKjbuic?si=ViHFmlzm7g3CA-2v

Bottom Up Parsers: https://youtu.be/SemmXpNeTx4?si=exAXZM6Wxt1_4DQ8
https://youtu.be/v8p2BrEUFbc?si=zVz_tK9h6rumAxne
Introduction to LR Parsing, L-R Parsing Algorithm,Viable Prefixes:Reduce in final closure state all terminal
https://youtu.be/qTJR9Jw_i2I?si=RWuErY6gzCkFps1E
Simple LR Parser (SLR), Construction of Simple LR Parsing Table: reduce in left-hand’s follow terminal
https://youtu.be/p7gsXHibb_w?si=4JyZw9EPFC4Wxb57
Canonical LR(1), Construction of LR(1) Parsing Table : https://youtu.be/hLycx5wiU8g?si=L75NI-L1Lbxwu9-Z
Look Ahead LR (LALR), Construction of LALR Parsing Table: https://youtu.be/0YsmwW7tmjo?si=P2vMtaZ-jiFK87UE

Question Bank Remaining Questions CH.3:

Find FOLLOW for given grammar S → ACB / CbB / Ba, A → da / BC, B → g / ∈, C → h / ∈
First(S) = { d , g , h , ∈ , b , a }
First(A) ={ d , g , h , ∈ }
First(B) = { g , ∈ }
First(C) = { h , ∈ }
Follow(S) = { $ }
Follow(A) = { h , g , $ }
Follow(B) = { $, a , h , g }
Follow(C) = { g , $, b , h }

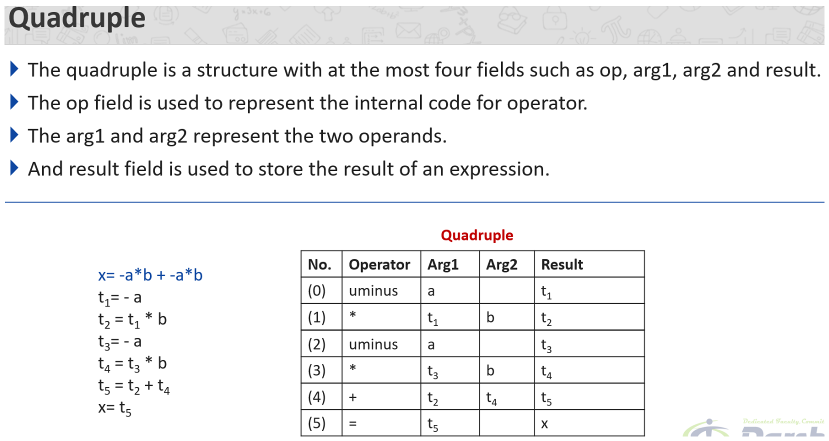
Question Bank Chapter 3 Sum Answer:
https://drive.google.com/file/d/11_rOVhhEhmjnLblPvNDDkHUixDqbuMGh/view?usp=sharing

CHAPTER:4:

Syntax-Directed Definitions & Application of Syntax Directed Translation:
https://youtu.be/_CloXDbYAAg?si=m1nC9WXHsYtmPN8T
https://youtu.be/7wlyOkz61ig?si=NVSfTbc3T6gufnBi
How to Evaluate Arithmetic Expression using SDT: https://youtu.be/CrLGZlvNvCw?si=eIT7sCaoRWb8mrx5
S-attributed Definitions & L-attributed Definitions: https://youtu.be/w03voSY4REs?si=S1_DH6a-E2L872T2
Question on S-Attributed and L-Attributed: https://youtu.be/sY1pyew6ZJI?si=qMlmmauf6z0hyCA-
SDT to convert infix to postfix expression: https://youtu.be/blH8Yqinq7M?si=GMyVkY-OOiKLzBve
SDD to convert Binary to decimal: https://youtu.be/bNPX_F0x6Hc?si=2L4DFV3ApuO304Vn
SDT to produce Three Address Code: https://youtu.be/T0vWu3s2CsE?si=zi3U7D_NcS1kPbQZ
Infix to postfix conversion: https://youtu.be/41FCbf7IbUY?si=lDUiQf9EtWV4Lq_J (Using Manual Method)
https://youtu.be/N1Sr-j-cf4c?si=e8EsDC-UlCGzV-L5 (Using Stack)
Intermediate Code Generation: https://youtu.be/j-bLeUysUiE?si=KL-20fJCsm9ZIYEo
https://youtu.be/yMUw2MxzNec?si=Qgas1MZsYO6qifIV
Three Address Code*(ASKED IN UNIT TEST-2)*:
Representation/Implementation of three address code: https://youtu.be/_HGWcUwfNX4?si=ZtUlJye4X74pIoWl

- There are three types of representation used for three address code:
  1. Quadruples
  2. Triples
  3. Indirect triples
- Ex: $x= -a*b + -a*b$

$$\left.\begin{array}{l} t_1 = -a \\ t_2 = t_1 * b \\ t_3 = -a \\ t_4 = t_3 * b \\ t_5 = t_2 + t_4 \\ x = t_5 \end{array}\right\} \text{Three Address Code}$$



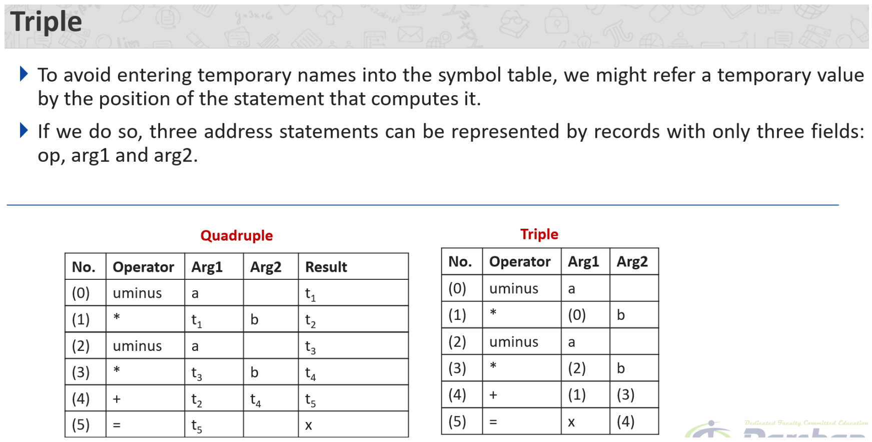
Quadruple

- The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- The op field is used to represent the internal code for operator.
- The arg1 and arg2 represent the two operands.
- And result field is used to store the result of an expression.

$x= -a*b + -a*b$
$t_1 = -a$
$t_2 = t_1 * b$
$t_3 = -a$
$t_4 = t_3 * b$
$t_5 = t_2 + t_4$
$x = t_5$

Quadruple

No.	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x



Triple

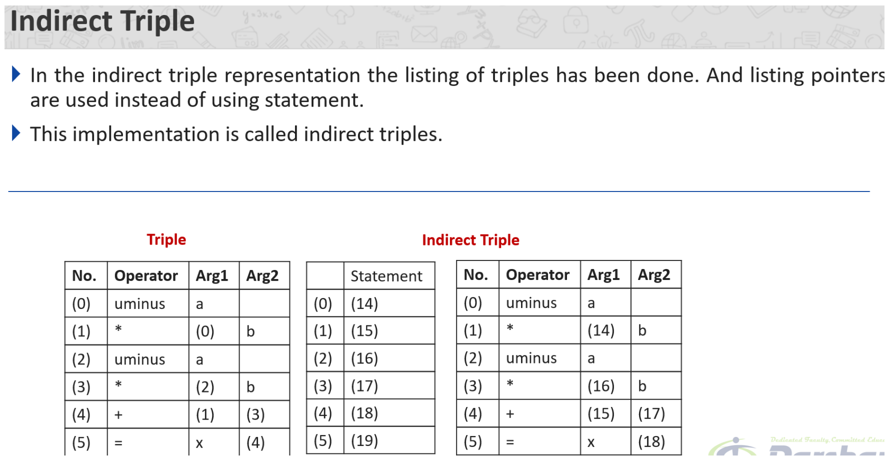
- To avoid entering temporary names into the symbol table, we might refer a temporary value by the position of the statement that computes it.
- If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.

Quadruple

No.	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Triple

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)



Indirect Triple

- In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- This implementation is called indirect triples.

Triple

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect Triple

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(14)	b
(2)	uminus	a	
(3)	*	(16)	b
(4)	+	(15)	(17)
(5)	=	x	(18)

Variants of Syntax Trees: https://youtu.be/lK9kEmdeqF0?si=noWfiGf5iRsv-4Tk

Notes of above video: https://www.gatevidyalay.com/syntax-trees/

Three address code Examples: https://youtu.be/zvf1sWcCtTg?si=dNlNSj68a0-xvunN

Notes of above video: https://www.gatevidyalay.com/three-address-code/

Control Flow & Basic Block and Flow Graph: https://youtu.be/fwJnIqv80jc?si=_S7GnZtK6F0avHuu

More Questions on Basic Block and Flow Graph: https://youtu.be/tKnNqiU4gCI?si=bKyr1d0s_0q2H4gv
https://youtu.be/tkSsh91ehUA?si=lMpoXQXlNJ97W-QY

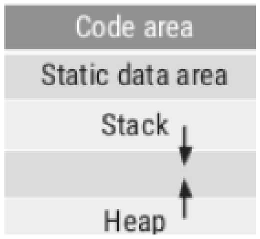
Directed Acyclic Graphs (DAGs): https://youtu.be/FOZInDh8Ym0?si=2rNPJtshgpGOGr5j

Notes of above Video: https://www.gatevidyalay.com/directed-acyclic-graphs/

CHAPTER:5:

Storage Organization:

0The compiler demands for a block of memory to operating system. The compiler utilizes this block of memory executing the compiled program. This block of memory is called run time storage.



The run time storage is subdivided to hold code and data such as, the generated target code and data objects. The size of generated code is fixed. Hence the target code occupies the determined area of the memory. Stack is used to manage the active procedure.Heap area is the area of run time storage in which the other information is stored.

Activation Tree & record: https://youtu.be/DtnkpvY8LwM?si=j40z7vzzhtoxP5zs

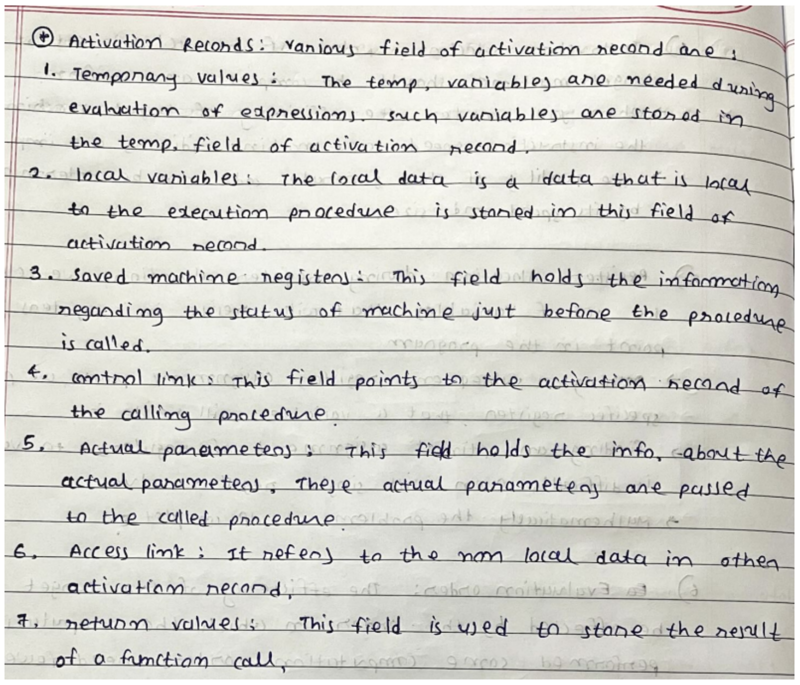
An activation tree, in the context of compiler design, is a data structure that represents the runtime organization of procedures or functions in a program. It is also known as a call tree or call graph. The activation tree is essential for managing the execution of procedures and functions, as well as for handling local variables, parameters, and the control flow during program execution.

Here's how it works:

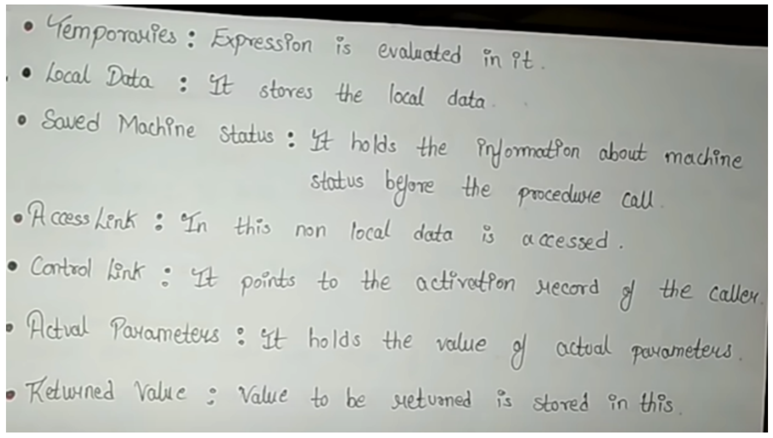
Activation Records (AR): Each procedure or function call in a program creates an activation record. An activation record, sometimes called a stack frame, contains information related to that specific call, such as the function's local variables, parameters, and the return address.

Nesting: Activation records are organized in a nested manner. When one function calls another function, a new activation record is created and linked to the calling function's activation record.

Activation Tree: The activation tree is a visual representation of this nested structure. Each node in the tree represents an activation record, and the links between nodes represent the calling relationships between functions.



⊕ Activation Records: various field of activation record are :
1. Temporary values : The temp. variables are needed during evaluation of expressions. such variables are stored in the temp. field of activation record.
2. local variables : The local data is a data that is local to the execution procedure is stored in this field of activation record.
3. saved machine registers : This field holds the information regarding the status of machine just before the procedure is called.
4. control link : This field points to the activation record of the calling procedure.
5. Actual parameters : This field holds the info. about the actual parameters. These actual parameters are passed to the called procedure.
6. Access link : It refers to the non local data in other activation record.
7. return values : This field is used to store the result of a function call.



- Temporaries : Expression is evaluated in it.
- Local Data : It stores the local data.
- Saved Machine Status : It holds the information about machine status before the procedure call.
- Access Link : In this non local data is accessed.
- Control Link : It points to the activation record of the caller.
- Actual Parameters : It holds the value of actual parameters.
- Returned Value : Value to be returned is stored in this.

Storage Allocation Strategies:
There are mainly three types of Storage Allocation Strategies:
1. Static Allocation 2. Heap Allocation 3. Stack Allocation
Static Allocation: In static allocation, memory is allocated at compile-time, and the allocation remains fixed throughout the program's execution. Variables have a fixed memory location and size. It is simple and efficient but does not support dynamic data structures or recursive function calls. C and C++ use static allocation.
<u>Advantages of Static Allocation:</u>
It is easy to understand.
The memory is allocated once only at compile time and remains the same throughout the program completion.
Memory allocation is done before the program starts taking memory only on compile time.
<u>Disadvantages of Static Allocation:</u>
Not highly scalable.
Static storage allocation is not very efficient.
The size of the data must be known at the compile time.
Stack Allocation: Stack is commonly known as Dynamic allocation. Dynamic allocation means the allocation of memory at run-time. Stack is a data structure that follows the LIFO principle so whenever there is multiple activation record created it will be pushed or popped in the stack as activations begin and ends.
When the activation record gets popped out, the local variable values get erased because the storage allocated for the activation record is removed. C and C++ both have support for Stack allocation.
<u>Advantages</u>:
Memory Deallocation is also fast in stack-based allocation.
Function calls can be efficient using stack-based allocation.
<u>Disadvantages</u>:
The amount of memory is limited for stack-based allocation.
As the size of the stack is limited, issues like stack overflow can happen.
Heap Allocation: Heap allocation is used where the Stack allocation lacks if we want to retain the values of the local variable after the activation record ends, which we cannot do in stack allocation
Heap allocation is used for dynamic data structures such as linked lists, trees, and dynamic arrays.
Heap is the most flexible storage allocation strategy we can dynamically allocate and de-allocate local variables whenever the user wants according to the user needs at run-time. C, C++, Python, and Java all of these support Heap Allocation.
The programmer must manage memory deallocation (e.g., using malloc and free in C).
<u>Advantages of Heap Allocation:</u>
Heap allocation is useful when we have data whose size is not fixed and can change during the run time.
We can retain the values of variables even if the activation records end.

Heap allocation is the most flexible allocation scheme.
Disadvantages of Heap Allocation:
Heap allocation is slower as compared to stack allocation.
It allows for flexible memory management but can lead to memory leaks and fragmentation if not managed carefully.

Issues in Code Generator*(ASKED IN UNIT TEST-2)*
1. Input to the code generator: The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code
The code generation phase needs complete error-free intermediate code as an input requires.
2. Target program: The target program is the output of the code generator. The output can be:
Assembly language, Relocatable machine language, Absolute machine language but that should be compatible with target architecture.
3. Memory management: Memory management in code generation includes deciding where and how to allocate memory for variables and data structures. This involves stack and heap memory management, allocation and deallocation, and addressing modes. Dealing with issues like memory leaks and fragmentation is essential.
4. Instruction selection: Deciding which machine instructions to use for different high-level language constructs is a non-trivial task. The code generator needs to map abstract operations to the most efficient and suitable machine instructions, taking into account the target architecture's capabilities.
5. Register Allocation: Efficient register allocation is a critical challenge. The code generator must decide which variables to keep in registers, when to spill variables to memory, and how to manage the limited number of registers available on the target machine. Register allocation strategies impact code performance.
6. Evaluation order: The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

Peephole Optimization*(ASKED IN UNIT TEST-2)*: https://youtu.be/clb4tnEm8l4?si=guhscfEDVxKH6-ql
Peephole optimization is a type of code Optimization performed on a small part of the code. It is performed on a very small set of instructions in a segment of code.
The small set of instructions or small part of code on which peephole optimization is performed is known as peephole or window.
It basically works on the theory of replacement in which a part of code is replaced by shorter and faster code without a change in output. The peephole is machine-dependent optimization.
Objectives of Peephole Optimization:
1. To improve performance
2. To reduce memory footprint
3. To reduce code size
Peephole Optimization Techniques:
Redundant load and store elimination: In this technique, redundancy is eliminated.
Initial code:
y = x + 5;
i = y;
z = i;
w = z * 3;
Optimized code:
y = x + 5;
w = y * 3;
Constant folding: The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.
Initial code: x = 2 * 3;
Optimized code: x = 6;
Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.

Initial code: y = x * 2;
Optimized code: y = x + x;
Null sequences/ Simplify Algebraic Expressions : Useless operations are deleted.
Initial Code: a = a+0; b = b*1;
Optimized: a = a; b = b;
Deadcode Elimination: In this technique, As the name suggests, it involves eliminating the dead code.
The statements of the code which either never executes or are unreachable or their output is never used are eliminated.
Initial Code:

```
i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
```

Optimized Code:

```
i = 0;
```

QuestionBank Remaining Questions CH.5:

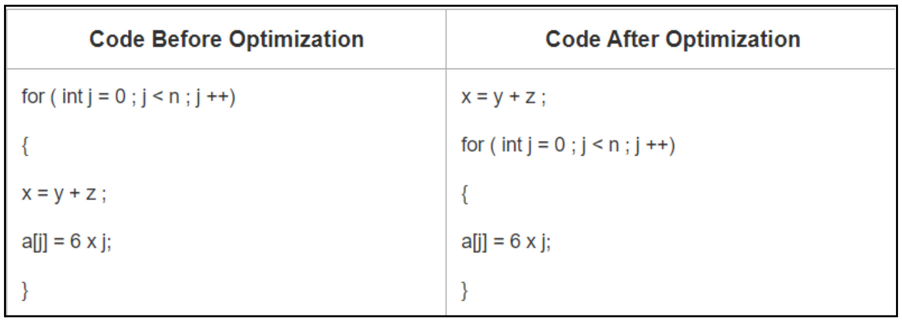
Explain Dynamic storage allocation technique: That means Stack allocation and Heap Allocation from Storage Allocation Strategies.

Discuss any three methods for code optimization.
Important code optimization techniques are-
1. Compile Time Evaluation- Two techniques that falls under compile time evaluation are-
A) Constant folding: The code that can be simplified by the user itself, is simplified. Here simplification to be done at runtime are replaced with simplified code to avoid additional computation.
Initial code: x = 2 * 3;
Optimized code: x = 6;
B) Constant Propagation- In this technique, If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
Example:
pi = 3.14
radius = 10
Area of circle = pi x radius x radius

- This technique substitutes the value of variables ‘pi’ and ‘radius’ at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves the time at run time.

2. Code Movement-
In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.



Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

3. Deadcode Elimination: In this technique, As the name suggests, it involves eliminating the dead code.
The statements of the code which either never executes or are unreachable or their output is never used are eliminated.
Initial Code:

```
i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
```

Optimized Code:

```
i = 0;
```

4. Strength Reduction: The operators that consume higher execution time are replaced by the operators consuming less execution time.
Initial code: y = x * 2;
Optimized code: y = x + x;

Explain the calling sequence with an example.
Todo

UNIT TEST EXAM PAPER:

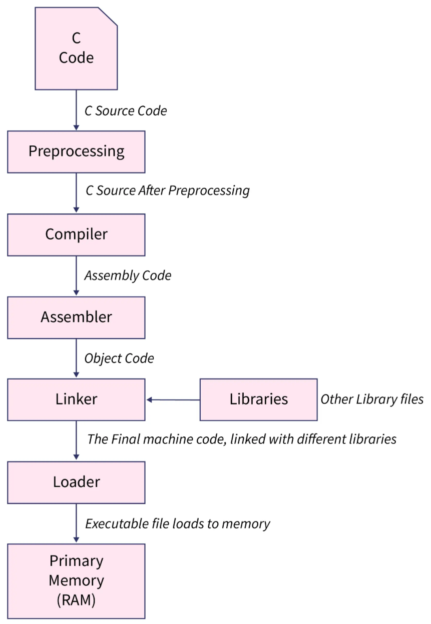
Unit Test 1: https://drive.google.com/file/d/1XAdOZ4EI8izATVJuNMZYCAa-42ud33zB/view?usp=sharing
Unit Test 2: https://drive.google.com/file/d/1zvHdgeBq5o6B3pWke8gOE7NbFR4jFITI/view?usp=sharing
Remaining Question's Answer:
Q. Differentiate lexical and syntax error with appropriate example.
Lexical Error:
Definition: Lexical errors, also known as lexical analysis errors or scanning errors, occur when the compiler encounters illegal characters or sequences of characters that do not conform to the language's lexical rules. Lexical analysis is the first phase of compilation, and it deals with breaking the source code into tokens.
Example: Consider a simple programming language where variables must start with a letter followed by letters or digits. If the source code contains a variable name like "123var," the compiler would report a lexical error because it starts with a digit, violating the lexical rules.

```
# Lexical Error Example
123var = 42
```

Syntax Error:
Definition: Syntax errors occur when the compiler encounters code that violates the grammatical rules of the programming language. The syntax phase of compilation involves analyzing the structure of the code based on the language's grammar rules.
Example: In many programming languages, a common syntax rule is that statements must end with a semicolon. If a semicolon is missing at the end of a statement, the compiler would report a syntax error.

```
// Syntax Error Example
public class Example {
    public static void main(String[] args) {
        System.out.println("Hello, World")  // Missing semicolon
    }
}
```

Q. Explain How a C program is executed in Compiler Design.



C code: When you first write the code in the C language, that source code is sent to the Preprocessing section.
Preprocessing: In this section our source code is attached to the preprocessor file. Different types of header files are used like the studio.h, math.h, etc. Our C source code is attached to these types of files and the final C Source generates. (some of the preprocessor directives are #include,#define). After generating the preprocessed C source code, the C source code is sent to the compiler section.
Compiler − After generating the preprocessed source code, it moves to the compiler and the compiler generates the assembly level code after compiling the whole program.
The first thing the C compiler does is to search for any error. If there is no error, the C compiler will report for no error, after that the compiler will store the file as a .obj file of the same name, which is termed as the object file. Although this .obj file will not be executable. After the compilation, the process is continued by the assembler section.
Assembler: This part usually generates the Object code, after taking the assembly-level code from the compiler. This object code is quite similar to the machine code or the set of binary digits. After this assembler part, The Linker continues the process, producing an executable.exe file at the end.
Linker − Linker is another important part of the compilation process. It takes the object code and link it with other library files, these library files are not the part of our code, but it helps to execute the total program. After linking the Linker generates the final Machine code which is ready to execute.
Loader − A program, will not be executed until it is not loaded in to Primary memory. Loader helps to load the machine code to RAM and helps to execute it. While executing the program is named as Process. So process is (Program in execution).

Q. Discuss synthesized attributes and inherited attributes in details.
Synthesized Attributes
These attributes get values from the attribute values of their child nodes. To illustrate, assume the following production:
S → ABC
If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
Synthesized attributes never take values from their parent nodes or any sibling nodes.
Example: E → E + T {E.value = E.value + T.value}
So here the parent node E gets its value from its child node.
Inherited Attributes
In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. As in the following production,
S → ABC
A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.
Example: E → T {T.value = E.value}
So here the child node T gets its value from its parent node E.

ASSIGNMENTS:

Assignment: https://drive.google.com/file/d/18BGeQEGefJu4yrqnXZjFQR3hB1-3dPdB/view?usp=sharing

Solution: https://drive.google.com/file/d/1nhBoOBbVLpqzQkFk5EKgyx2LpIEHRlqq/view?usp=sharing

CSPIT Assignment:

Assignment 1&2: https://drive.google.com/file/d/1G2GGoA3hyhHyY-Lf7he8yPZ-fzafOW9n/view?usp=sharing

Assignment 3: https://drive.google.com/file/d/11r3cjVZDJp9LpmTp4Vb7jfbSkiflnU0K/view?usp=sharing