compiler is a softw. prog. that converts the high level lang. or source code into low level lang. or machine code that can be understood by the comput. or machine. The proc. of converting the source code into machine code involves phases: 1. Lexical analysis: The 1st phase of compiler. This phase reads the code & breaks it into a stream of tokens. The tokens are then passed to next phase for further proc.ing 2. Syntax analysis: The 2nd phase of compl. This phase takes the stream of tokens generated by lex. ana. & checks whether the tokens are in grammar of prog. lang. The o/p of this phase is a syntax tree. 3. Semantic analysis: The 3rd... this phase checks whether the code is semantically correct. 4. Interm. code generation: The 4th... This phase generates an interm. representation of the source code that can be easily translated into machine code. 5. Code optimization: ... This phase applies various optimization techq.s to interm. code to improve performance of the generated machine code. 6. Code generation: ... This phase takes optimized interm. code & generates actual machine code that can be executed by machine.

In a Lang. proc.ing sys. the source code is 1st preprocessed. The modified source code is compiled & is proc.ed by the compl. to form the target assembly lang. which is then translated by the assembler to create relocatable obj. code that are proc.ed by linker & loader to create the target prog. or machine code. Bootstrapping is a proc. in which simple lang. is used to translate more complicated prog. which in turn may handle for more complicated prog. This complicated prog can further handle more complic. prog. & so on... It is also defined as self compiling compiler written in its source lang. Porting is the proc. of modifying the existing compiler for the target env to work on a diff. machine. I/P buffering is a techq. involves reading blocks of i/p chars from source code & grouping them into meaningful units called tokens. I/P buffering improves performance & reduce overhead.

Specification of Token: There are 3 specifications of token: 1. Strings: are a finite set of symbols or chars. These symbols can be a digit or an alphabet. 2. Langs. can be defined as a finite set of strings over some symbols or alphabets. 3. Regular Expressions are str.s of chars that define a searching pattern with which we can form a lang. & each RE represents a lang. A RE 'r' can denote a lang L(r) which can be built recursively over the smaller RE by following rules → Kleene closure (*), positive closure (+)

Recognition of Tokens: Tokens are recognized using Lexemes → digit → [0-9] number → digit | E(+|-)? digit, letter → [A-Z a-z], id → letter (letter | digit)* if → if, then → then, else → else, relop → < | > | <= | >= | = | <>

Syntax analysis also known as parsing is a proc. where the compl. checks if the source code follows the grammatical rules of prog. lang. The main goal of syn. ana. is to create a parse tree of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the prog. There are some types of parsing algs used in syn. ana. → LL parsing: This is a top down parsing alg. that starts with the root of parse tree & constructs the tree by successively expanding non-terminals. LL parsing is simple & easy to implement. LR parse: This is a bottom up parsing alg. that starts with the leaves of the parse tree & constructs the tree by successively reducing terminals. LR par. is more powerful than LL par. & can handle a larger class of grammars. LR(1) par.: This is a variant of LR par. that uses lookahead to disambiguate the grammar. LALR par. is a variant of LR par. that uses a reduced set of lookahead symbols to reduce the no. of states in the LR par se.

Top down parsing: The proc. of constructing the parse tree which starts from the root & goes down to the leaf is Top down par. Top down parsers constructs from the grammar which is free from ambiguity & left recursion. → top down parsers use leftmost derivation to construct a parse tree. Bottom up parsing starts from the leaf nodes of a tree & works in upward direction till it reaches the root node. Here, we start from a sentence & then apply production rules in reverse manner in order to reach the start symbol.

YACC - Yet Another Compiler compiler:
→ It provides a tool to produce a parser for a given grammar. → YACC is a prog. designed to compile a LALR(1) grammar → It is used to produce the source code of the syntactic analyzer of the lang. produced by LALR(1) grammar → the i/p of YACC is the rule of grammar & the o/p is a C prog. YACC i/p file is divided into 3 parts: i. defn. & Rules & Auxil. routines

The tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree. Evaluation order for SDD includes how the SDD (Syn. Directed Defn.) is evaluated with the help of att.s, dependency graphs, semantic rules & S and L attributed defn. SDD helps in the semantic analysis in the compiler. It is important to know how SDDs are evaluated & their evaluation order. 1. Dependency graphs: Dep. graph provides info. the order of eval. of att.s with the help of edges. An edge from the 1st node att. to the 2nd node att. gives the info. that 1st node att. eval. is required for the eval. of 2nd node att. Edges represents the semantic rules of the corresponding production. Dep. graph rules → A node in the dep. graph corresponds to the node of the parse tree for each att. → Edges of the dep. graph represent that the att. of 1st node is evaluated before the att. of 2nd node. 2. Ordering the Evaln. of att.s: The dep. graph provides admissible orders

Syntax Directed Definition (SDD): specifies the value of attributes by associating semantic rules with the grammar. It is a generalization of context free grammar in which each grammar production $X \to \alpha$ is associated with it a set of production rules of the form $s = f(b_1, b_2, b_3 \dots b_k)$ where s is the att. obtained from func. f. The att. can be str., number, type or a mem. loc.
Eg: $E \to E_1 + T \Rightarrow \{E.val = E_1.val + T.val\}$

YACC format: input-file.y → [YACC] → y.tab.c
y.tab.c → [C comp.] → a.out
a input → [a.out] → o/p (Syntax Directed Translation: has augmented rules to the grammar that facilitate semantic analysis. SDT involves passing info. bottom up or top down to the parse tree in form of att.s attached to the nodes. SDT rules use lexical values of nodes, constants, att.s associated with non terminals in their defn. → The general approach of SDT is to construct a parse tree or syntax tree & compute the values of att.s at the nodes of the tree

The eval. order att.s of the nodes of the parse tree. An edge in the dep. graph represents that the att. of the 2nd node is dependent on the att. of the 1st node for further eval. This order of evaluation gives a linear order called topological order. → We can't evaluate SDD on a parse tree when there is a cycle present in the graph & due to the cycle, no topological order exists. S Attributed defn. S-att.ed SDD can have only synthesized att.s. In this type of defn. semantic rules are placed at the end of the production only. Its eval. is based on bottom up par. Eg: $S \to AB\{S.x = f(A.x, B.x)\}$. L Attributed Defn.: can have both synthesized & inherited. In this type of defn., semantic rules can be place anywhere in the RHS of the production. Its eval. is based on inorder. Eg: $S \to AB\{A.x = S.x + 2\}$ or $S \to AB\{B.x = f(A.x, B.x)\}$. Semantic rules with controlled side effects: Side effects are the prog. fragment contained within semantic rules. These side effects in SDD can be controlled in 2 ways: permit only benign side effects & constraints to have the same transl.

Activation tree is a tree struct. that represents func. calls made by a prog. during a execution. Activation records Proce dure calls & returns are usually manag ed by a runtime stack called the control stack. Each live activation has an activati on record. Info. needed for each exec ution of a procedure is stored in a reco rd called activation record. Return values | Actual param.s | control link | Acces s link | saved machine status | local data | Te mporaries | Storage organization Compiler demands a block of mem. from OS to run compiled prog. This block is called as runtime storage. Run time storage comes in blocks of contiguous bytes. It is divided into parts to hold code & data.

Code | static data | stack | → free mem. | Heap |

→ Mem. loc. for code are determin at compile time → Locs of static da ta can also be determined at compile time
→ Stack - Data obj.s allocated at run time
→ Heap - other dynamically allocated data objs at run time (For eg. Malloc area in C)

Parameter passing refers to the exchan ge of info b/w methods, func.s & procedu res. Some mechanism transfers the val ues from a variable procedure to the called procedure. Parameters are 2 types: 1. Actual params are variables that accept data given by the calling proc. These variables are listed in the calling func's definition. They accept calling process's data. 2. Formal param. are variables to hose values & func.s are given to the ca lled func. These variables are supplied as param.s in the func. call.

R value refers to value contained in the mem or storage. They are basically placed on right side of the assignment operator Eg: a=1; b=a+7 → R values are 1, a+7

L value of the expr. refers to the loc. in mem. or storage where the expr. is kept Eg: x=10; a=20 → L values: x, a; R: 10, 20

Methods: 1. call by value: R values are passed to formals. During this, the compl. adds the r value of the actual param.s that were passed to the calling procedure, to the called procedu re's activation record. Eg: case 2 call by L-values are passed to formals. Eg: code swap(*p, *q) & that t, .... 3 call by copy R values are passed to formals before the call L values are determined. 4. call by name Actual params are substituted for the formals. Heap management: The heap is the portion of the storage that is used for data that lives indefinitely or until the prog. explicitly deletes it. Garbage Data that can't be referenced is gener ally known as garbage. Garbage collection is the prog. of automatically recovering unused computer memory storage. Automatic garb. coll. is used in many high-level lang's

Storage allocation techniques: 1. Stack storage alloc.: If the size of data objects is known at compile time, names are bound to sto rage at compile time only. Hence every time procedure is invoked, its names are bound to the same storage alloc only. Mem. allocated for objs won't be changed in run time, hence this alloc is called static. In this, it is easy for the com piler to find the addresses of the objs in activation record. stack alloc: Stack is used for sto. alloc. This stack is called as control stack. Almost all lang.s have procedures, func.s or methods as units of user defined actions. Each time a func. is called, space for its local variables is pushed onto a stack, & when the procedure terminates, that space is popped out of the stack. Heap alloc: If non local variables must be retained even after the activ. record, then heap alloc. is required which is not possible in stack alloc. Heap all. allocates contin uous blocks of mem when req.ed for sto rage of activ. records or for other data objs. It can be deallocated when activ. ends, & it can be reused by heap manager. Linked

**Code generator** converts the interm representation of source code into a form that can be readily executed by the machine. Designing of the code generator should be done in such a way that it can be easily implemented, tested & maintained. → The following are issues arises during the code gen. phase: 1. I/p to code generator. → Target prog. → mem. management 2. Instruction selection 3. Register alloc. Issues. Target prog. is the o/p of the code gen. The o/p may be absolute machine lang. / relocatable machine lang. or assembly lang. Abs. machine lang. its o/p has the advantages that it can be placed in a fixed mem. loc. & can be immediately executed. **Basic block** is a seq. of consecutive stmts in which flow of ctrl. enters at the beg. of the block & leaves at the end without halt or possibility of branching. **Flowgraph** Once an interm. code prog. is partitioned into basic blocks, we represent the flow of ctrl b/w them by a graph.

**Principal sources of optm.** 1. **common sub expression elimination**: An occurance of an expr. E again is called common sub expr. if E was previously computed, & the values of the variables in E have not changed. we can avoid recomputing the expr. if we can use the previously computed value. Eg| 2. **Copy propagation**: Assignments of the form f:=g called copy stmts. or copies. The idea behind copy propag. transfm is to use g for f, whenever possible after the copy stmt. copy propag. means use of one variable instead of another. 3. Dead code elim. 4. **Constant folding**: The code that can be simplified by the user itself, is simplified. Here, simplification to be done at runtime are with simplified code to avoid additional computation. **Peephole optm.** A simple but effective tech. for improving the target code is peephole optm.: a method for trying to improving the perfm. of the target prog. by examining a short

**optimization of Basic Blocks**: optm. is applied to the basic blocks after the interm. code generation phase of the compiler. Optm. is the proc. of transforming a prog. that improves the code by consuming fewer resources & delivering high speed. In optm, high-level codes are replaced by their equivalent low-level codes. There are 2 types of basic block optm.: 1. Structure preserving transfms 2. Algebraic transfms. 1. → a) dead code elimination b) common subexpression elimination c) renaming of temp. variables d) Interchange of 2 indep. adjacent stmts 2 → a) constant folding b) copy propagation c) Strength reduction

**Dead code elimination**: Dead code is defined as the part of the code that never executes during the prog. execution. So for optm., such code or dead code is eliminated. The code which is never executed during the prog. (dead code) takes time. So, for optm. i.e. speed, it is eliminated from the code. Eliminating the dead code increases the speed of the prog. as compiler doesn't have to translate the dead code.

seq. of target instruction (called peephole) & replacing these instructions by a shorter or faster seq. whenever possible. objectives of peephole optm. 1. To improve perfm. 2. To reduce mem. footprint 3. To reduce code size. **Local optm**: Transfms are applied to small blocks of statements. **Global optm**: Transfms are applied to large prog. segments that includes func.s, procedures & loops. → copy propag. → code motion → Dead code eliminations.

Symbol table is an important data structure created & maintained by compl.s in order to store info. about the occurence of various entities such as variable names, func. names, objs, cls.s, interfaces etc. Symbol table is used by both the analysis & the synthesis parts of a compl. <Symbol name, type, attr>

Implementation If a compl. is to handle a small amount of, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. Symbol table can be implemented in the following ways.

1. Linear list 2. Binary search trees. Hash table. 1. List: We use a single arr. or equivalently several arr.s to store names & their associated info. New names are added to the list in the order in which they are encountered. The position of end of arr. is marked by the pointer available, pointing to where the next symbol entry will go. The implementation using a linked list, a link field is added to each record.

Operations on Symbol Table 1. Insert() op. is more frequently used in the analysis phase. Insert() op. is used to insert the info in the symbol table like the unq. name occurring in the source code. 2. Lookup: In the sym. tab, lookup() op. is used to search a name. It is used to determine: → the existence of symb. in the tab. → the declarations of the symb. before it is used. Intermediate code generator receives i/p from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation eg. postfix notation. Interm. code tends to be machine indep. code. ∴ code generator assume to have unlimited no. of mem. space to generate code. Eg:- a = b+c*d ; 3 addr code r1 = c*d; r2 = b+r1; a = r2. Interm. code supports eliminating the reqm of a new compiler for every individual machine by upholding the same analysis part for all the compl.s.

3 addr. code is a type of interm. code which is easy to generate & can be easily converted to machine code. It makes use of at most 3 addr.s & one operator to represent an expr. & the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of op. given by 3 addr. code. → There are 3 representation of 3 addr. code: 1. Quadraple 2. Triples 3. Indirected Triples.

Translation of expressions: 1. op.s within exprs 2. Incremental transl. 3. Addr. of arr. elements 4. Transl. of arr. References 1. op.s within exprs Eg: S → id = E ⇒ scode = E.code || gen(top.get(id.lexeme) '=' E.addr) E → E1 + E2 ⇒ E.addr = new Temp() E.code = E1.code || E2.code || gen(E.addr '=' E1.addr '+' E2.addr)

Type checking is the proc. of verifying & enforcing constraints of types in values. A compl must check that the source prog. should follow the syntactic & semantic conventions of the source lang. & It should also check the type rules of the lang. It allows the prog. er to limit what types may be used in certain circumstances & assigns types to values. The type checker determines whether these values are used appropriately or not. Backpatching is the activity of filling up unspecified or missing info. of labels using appropriate semantic actions during the code generation phase. Need for Backpatching: 1. Boolean expr. 2. Flow of control statements.

Runtime env. is a state of the target machine, which may include softw. libs, env. variables etc to provide services to the process running in the system.