|  |  |
| --- | --- |
| **UNIT - I / PART B C6660.1** | |
| **1** | **What is a compiler? State various phases of a compiler and explain them in detail. (16)**  **COMPLIER:** Complier is a program that reads a program written in one language –the source language- and translates it into an equivalent program in another language- the target language.  In this translation process, the complier reports to its user the presence of the errors in the source program.  **The classifications of compiler:** Single-pass compiler, Multi-pass compiler, Load and go compiler, Debugging compiler, Optimizing compiler.  **INTERPRETER:** Interpreter is a language processor program that translates and executes source code directly, without compiling it to machine code.  There are two major parts of a compiler: Analysis and Synthesis  • **In analysis phase**, an intermediate representation is created from the given source program. – Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this  phase.  • **In synthesis phase**, the equivalent target program is created from this intermediate representation. – Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.  ***Phases of compiler***  Each phase transforms the source program from one representation into another representation.   * They communicate with error handlers. * They communicate with the symbol table.   **Lexical Analyzer**   1. Lexical Analyzer reads the source program character by character and returns the tokens of the source program. 2. A token describes a pattern of characters having same meaning in the source program.(such as identifiers, operators, keywords, numbers, delimeters and so on)   **Ex**  • Puts information about identifiers into the symbol table.  • Regular expressions are used to describe tokens (lexical constructs).  • A (Deterministic) Finite State Automaton can be used in the implementation of a lexical  analyzer.  **Syntax Analyzer**   1. A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. 2. A syntax analyzer is also called as a parser. 3. A parse tree describes a syntactic structure.   **Example:**  In a parse tree, all terminals are at leaves.  All inner nodes are non-terminals in a context free grammar.  **Semantic Analyzer**   1. It checks the source program for semantic errors and collects the type information for the subsequent code-generation phase. 2. It uses hierarchical structure determined by the syntax-analysis phase to identify the 3. operators and operands of expressions and statements. 4. An important component of semantic analysis is type checking. 5. Normally semantic information cannot be represented by a context-free language used in syntax analyzers. 6. Context-free grammars used in the syntax analysis are integrated with attributes   (semantic rules)  – the result is a syntax-directed translation,  – Attribute grammars  **Intermediate Code Generation**   1. A compiler may produce an explicit intermediate code representing the source program. 2. These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes. 3. An intermediate form called "three-address code” which is like the assembly   language for a machine in which every memory location can act like a register.   1. Three-address code consists of a sequence of instructions, each of which has at most   three operands.  **Code Optimizer**   1. The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space. 2. The code optimization phase attempts to improve the intermediate code, so that faster running machine code will result.  * Optimization may involve: * Detection and removal of dead(unreachable) code * Calculation of constant expressions and terms * Collapsing of repeated expressions into temporary storage * Loop controlling * Moving code outside of loops * Removal of unnecessary temporary variables.   **Code Generator**  Produces the target language in a specific architecture.  The target program is normally is a re-locatable object file containing the machine codes.  This phase involves:   * Allocation of registers and memory * Generation of correct references * Generation of correct types * Generation of machine code.   **Symbol table**   1. An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier. 2. A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. 3. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. 4. When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table. 5. The attributes of an identifier cannot normally be determined during lexical analysis.   **For example**, in a Pascal declaration like var position\* initial, rate : real ;  The type real is not known when position, initial, and rate are seen by the lexical analyzer.  6. The remaining phases enter information about identifiers into the symbol table and  then use this information in various ways.  **Error Detection and Reporting**   1. Each compiler phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected. 2. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. 3. Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase. 4. During semantic analysis, the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. |
| **2** | **Explain the various phases of a compiler in detail. Also write down the output for the**  **following expression after each phase a:= b\*c-d.(16)** |
| **3** | **Write in detail about the analysis cousins of the compiler.(16) *(May 2013)***  **The cousins of a compiler are**   1. **Preprocessor** 2. **Assembler** 3. **Loader and Link-editor**   **PREPROCESSOR** A **preprocessor** is a program that processes its input data to produce output that is used as input to another program. The output is said to be a **preprocessed** form of the input data, which is often used by some subsequent programs like compilers. The preprocessor is executed before the actual compilation of code begins, therefore the preprocessor digests all these directives before any code is generated by the statements.  They may perform the following functions  1. Macro processing  2. File Inclusion  3. Rational Preprocessors  4. Language extension  **1. Macro processing:**   * A **macro** is a rule or pattern that specifies how a certain input sequence (often a sequence of characters) should be mapped to an output sequence (also often a sequence of characters) according to a defined procedure. * The mapping processes that instantiates (transforms) a macro into a specific output sequence is known as *macro expansion*.  macro definitions (#define, #undef) * To define preprocessor macros we can use #define.   **FORMAT:**  #define identifier replacement  int table2[100];  **2.File Inclusion:**   * Preprocessor includes header files into the program text. When the preprocessor finds an #include directive it replaces it by the entire content of the specified file. * There are two ways to specify a file to be included:   #include "file"  #include <file>  **3. Rational Preprocessors:**  These processors augment older languages with more modern flow of control and data structuring facilities.  For example, such a preprocessor might provide the user with built-in macros for constructs like while-statements or if-statements, where none exist in the programming language itself.  **4. Language extension:**   * These processors attempt to add capabilities to the language by what amounts to built-in macros. * For example, the language equal is a database query language embedded in C. * Statements begging with ## are taken by the preprocessor to be database access statements unrelated to C and are translated into procedure calls on routines that perform the database access.   **ASSEMBLER**   * Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. * The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. * Most assemblers also include macro facilities for performing textual substitution- * Example to generate common short sequences of instructions as inline, instead of *called* subroutines, or even generate entire programs or program suites.   There are two types of assemblers based on how many passes through the source are needed to produce the executable program.   * **One-pass assemblers** go through the source code once and assumes that all symbols will be defined before any instruction that references them. * **Two-pass assemblers** create a table with all symbols and their values in the first pass, then use the table in a second pass to generate code. The assembler must at least be able to determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.   **LINKERS AND LOADERS**  A **linker** or **link editor** is a program that takes one or more objects generated by a compiler and combines them into a single executable program.  **Three tasks**  1. Searches the program to find library routines used by program, e.g. printf(), math routines.  2. Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references  3. Resolves references among files Loader  A **loader** is the part of an operating system that is responsible for loading programs, one of the essential stages in the process of starting a program.  Loading a program involves reading the contents of executable file, the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running.  Once loading is complete, the operating system starts the program by passing control to the loaded program code. |
| **4** | **(i)Describe how various phases could be combined as a pass in a compiler?(8)*(May 2008)***  Logically each phase is viewed as a separate program that reads input and produced output for the next phase. In practice, some phases are combined.  **Front and Back Ends**  Modern compilers contain two parts, each which is often subdivided. These two parts are the front end and back end.   * The front end consists of those phases, or parts of phases, that depends primarily on the source language and is largely independent of the target machine. * These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. * A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases. * The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. * In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol-table operations.   **Passes**   * Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. * In practice, there is great variation in the way the phases of a compiler are grouped into passes, so we prefer to organize our discussion of compiling around phases rather than passes. * It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass. * For example, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped into one pass. * If so, the token stream after lexical analysis may be translated directly into intermediate code.   **Reducing the Number of Passes**   * It is desirable to have relatively few passes, since it takes time to read and write intermediate files. * If we group several phases into one pass, we may be forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it. * The internal form of the program may be considerably larger than either the source program or the target program, so this space may not be a trivial matter. * For some phases, grouping into one pass presents few problems. * For example, as we mentioned above, the interface between the lexical and syntactic analyzers can often be limited to a single token.   **(ii) Compiler Construction Tools (8)**  The following is a list of some useful compiler-construction tools:  **1. Parser generators:** These produce syntax analyzers, normally from input that is based on a context-free grammar.  In early compilers, syntax analysis consumed not only a large fraction of the running time of a com- compiler, but a large fraction of the intellectual effort of writing a compiler.  This phase is now considered one of the easiest to implement. Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand.  **2. Scanner generators:** These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.  **3. Syntax-directed translation engines:** These produce collections of routines that walk the parse tree, generating intermediate code.  The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.  **4. Automatic code generators:** Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine.  The rules must include sufficient detail that we can handle the different possible access methods for data; e.g.variables may be in registers, in a fixed (static) location in memory, or may be allocated a position on a stack. The basic technique is "template matching”.  **5. Data-flow engines:** Much of the information needed to perform good code optimization involves "data-flow analysis," the gathering of information about how values are transmitted from one part of a program to each other part. Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered. |
| **5** | **For the following expression (April/May,2017)**  **Position:=initial+ rate\*60**  **Write down the output after each phase** |
| **6** | **Describe the following software tools i. Structure Editors ii. Pretty printers iii. Interpreters**   * ***Structure Editor:*** A structure editor takes as input a sequence of commands to build a source program. * The structure editor not only performs the text-creation and modification function of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program. * ***Pretty printer*:** A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible, for example comment, indentation. * ***Static checker:*** A static checker reads a program, analyzes it and attempts to discover potential bugs without running the program.   ***Interpreter:*** Instead of producing a target program as a translation, an interpreter performs the operation implied by the source program. |
| **7** | **Elaborate on grouping of phases in a compiler.**  Logically each phase is viewed as a separate program that reads input and produced output for the next phase. In practice, some phases are combined.  **Front and Back Ends**  Modern compilers contain two parts, each which is often subdivided. These two parts are the front end and back end.   * The **front end** consists of those phases, or parts of phases, that depends primarily on the source language and is largely independent of the target machine. * These normally include **lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code.** * A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases. * The **back end** includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. * In the back end, we find aspects of the code optimization phase, and we find **code generation, along with the necessary error handling** and symbol-table operations.   **Passes**   * Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. * It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass. * For example, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped into one pass. * If so, the token stream after lexical analysis may be translated directly into intermediate code. * **Reducing the Number of Passes** * It is desirable to have relatively few passes, since it takes time to read and write intermediate files. * If we group several phases into one pass, we may be forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it. * The internal form of the program may be considerably larger than either the source program or the target program, so this space may not be a trivial matter.   For some phases, grouping into one pass presents few problems. For example, as we mentioned above, the interface between the lexical and syntactic analyzers can often be limited to a single token. |
| **8** | **What is difference between a phase and pass of a compiler? Explain machine dependent and machine independent phase of compiler.**   * Phase and Pass are two terms used in the area of compilers. * A pass is a single time the compiler passes over (goes through) the sources code or some other representation of it. * Typically, most compilers have at least two phases called front end and back end, while they could be either one-pass or multi-pass. * Phase is used to classify compilers according to the construction, while pass is used to classify compilers according to how they operate |
| **9** | **Explain the various errors encountered in different phases of compiler( Nov/Dec 2016)**  **(May/June,2016)**   * A parser should be able to detect and report any error in the program. * It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. * Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process.   A program may have the following kinds of errors at various stages:   * **Lexical:** name of some identifier typed incorrectly * **Syntactical:** missing semicolon or unbalanced parenthesis * **Semantical:** incompatible value assignment * **Logical:** code not reachable, infinite loop   There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.  **Panic mode**   * When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. * This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.   **Statement mode**   * When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. * For example, inserting a missing semicolon, replacing comma with a semicolon etc. * Parser designers have to be careful here because one wrong correction may lead to an infinite loop.   **Error productions**   * Some common errors are known to the compiler designers that may occur in the code. * In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.   **Global correction**   * The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.   When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. |
| **10** | **Explain language processing system with neat diagram. *(May 2016)***  The high-level language is converted into binary language in various phases. A compiler is a program that converts high-level language to assembly language. Similarly, an assembler is a program that converts the assembly language to machine-level language.  **Steps to execute C compiler, in a host machine:**   * First writes a program in C language (high-level language). * The C compiler, compiles the program and translates it to assembly program (low-level language). * An assembler then translates the assembly program into machine code (object). * A linker tool is used to link all the parts of the program together for execution (executable machine code). * A loader loads all of them into memory and then the program is executed.   **1. Preprocessor**   * A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. * It deals with macro-processing, augmentation; file inclusion, language extension, etc.   **2. Interpreter**   * An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. * A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes. * In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. * If an error occurs, an interpreter stops execution and reports it, whereas a compiler reads the whole program even if it encounters several errors.   **3. Assembler**   * An assembler translates assembly language programs into machine code. * The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.   **4. Linker**   * Linker is a computer program that links and merges various object files together in order to make an executable file. * All these files might have been compiled by separate assemblers. * The major task of a linker is to search and locate referenced module/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction to have absolute references.   **5. Loader**   * Loader is a part of operating system and is responsible for loading executable files into memory and execute them. * It calculates the size of a program (instructions and data) and creates memory space for it. It initializes various registers to initiate execution.   **6. Cross-compiler**   * A compiler that runs on platform (A) and is capable of generating executable code for platform (B) is called a cross-compiler.  1. **Source-to-source Compiler**  * A compiler that takes the source code of one programming language and translates it into the source code of another programming language is called a source-to-source compile |
| **11** | **(i) Give the transition diagram to represent relational operators.**  **(ii) Give the REGULAR EXP FOR Unsigned integer is N+** |
|  | **UNIT - II / PART –B CS6660.4** |
| **1** | **Explain about input buffering.(8)**  **Input Buffering:**   * + Some efficiency issues concerned with the buffering of input.   + A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.   + Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.   + There are three general approaches to the implementation of a lexical analyser:  1. Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input. 2. Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input. 3. Write the lexical analyser in assembly language and explicitly manage the reading of input.   ***Buffer pairs*:**   * + Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. The scheme to be discussed:   + Consists a buffer divided into two N-character halves.      * N – Number of characters on one disk block, e.g., 1024 or 4096.   + Read N characters into each half of the buffer with one system read command.   + If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.   + Two pointers to the input buffer are maintained.   + The string of characters between two pointers is the current lexeme.   + Initially both pointers point to the first character of the next lexeme to be found.   + Forward pointer, scans ahead until a match for a pattern is found.   + Once the next lexeme is determined, the forward pointer is set to the character at its right end.   + If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.   + If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.   –     ***Disadvantage of this scheme*:**   * + This scheme works well most of the time, but the amount of lookahead is limited.   + This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.   + For example: DECLARE ( ARG1, ARG2, … , ARGn ) in PL/1 program;   + Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.   ***Sentinels*:**   * + In the previous scheme, must check each time the move forward pointer that have not moved off one half of the buffer. If it is done, then must reload the other half.   + Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.   + This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.   + The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).      * + In this, most of the time it performs only one test to see whether forward points to an eof.   + Only when it reach the end of the buffer half or eof, it performs more tests.   Since N input characters are encountered between eof’s, the average number of tests per input character is very close to 1. |
| **2** | **Explain in detail about the role of Lexical analyzer with the possible error recovery actions.(16) *(May 2013)***  Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences.  The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.  If the lexical analyzer finds a token invalid, it generates an error.  The lexical analyzer works closely with the syntax analyzer.  It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.  Up on receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.  Its secondary tasks are,  • One task is stripping out from the source program comments and white space is in the form of blank, tab, new line characters.  • Another task is correlating error messages from the compiler with the source program.  Sometimes lexical analyzer is divided in to cascade of two phases.  1) Scanning  2) lexical analysis.  The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations. |
| **3** | **Describe the specification of tokens and how to recognize the tokens (16) (May 2013)**  **Specification of Tokens**  An alphabet or a character class is a finite set of symbols. Typical examples of symbols are letters and characters.  The set {0, 1} is the binary alphabet. ASCII and EBCDIC are two examples of computer alphabets.  **Strings** A string over some alphabet is a finite sequence of symbol taken from that alphabet.  For example, banana is a sequence of six symbols (i.e., string of length six) taken from ASCII  computer alphabet. The empty string denoted by , is a special string with zero symbols (i.e., string length is 0).  If x and y are two strings, then the concatenation of x and y, written xy, is the string formed by appending y to x.  For example, If x = dog and y = house, then xy = doghouse. For empty string, , we have S = S = S.  String exponentiation concatenates a string with itself a given number of times:  S2 = SS or S.S  S3 = SSS or S.S.S  S4 = SSSS or S.S.S.S and so on  By definition S0 is an empty string, , and S` = S. For example, if x =ba and na then xy2 = banana.  **Languages** A language is a set of strings over some fixed alphabet. The language may contain a finite or an infinite number of strings.  Let L and M be two languages  where L = {dog, ba, na} and M = {house, ba} then  Union: LUM = {dog, ba, na, house}  Concatenation: LM = {doghouse, dogba, bahouse, baba, nahouse, naba}  Expontentiation: L2 = LL  By definition: L0 ={} and L` = L  The kleene closure of language L, denoted by L\*, is "zero or more Concatenation of" L.  L\* = L0 U L` U L2 U L3 . . . U Ln . . .  For example, If  L = {a, b}, then  L\* = {, a, b, aa, ab, ab, ba, bb, aaa, aba, baa, . . . }  The positive closure of Language L, denoted by L+, is "one or more Concatenation of" L.  L+  = L` U L2 U L3 . . . U Ln  . . .  For example, If L = {a, b}, then  L+  = {a, b, aa, ba, bb, aaa, aba, . . . }  **Recognize the tokens**  **Finite Automata**   * A recognizer for a language is a program that takes a string x as an input and answers "yes" if x is a sentence of the language and "no" otherwise. * One can compile any regular expression into a recognizer by constructing a generalized transition diagram called a finite automaton. * A finite automation can be deterministic means that more than one transition out of a state may be possible on a same input symbol. * Both automata are capable of recognizing what regular expression can denote. * Nondeterministic Finite Automata (NFA) * A nondeterministic finite automaton is a mathematical model consists of * a set of states S; * a set of input symbol, ∑, called the input symbols alphabet. * a transition function move that maps state-symbol pairs to sets of states. * a state so called the initial or the start state. * a set of states F called the accepting or final state. * An NFA can be described by a transition graph (labeled graph) where the nodes are states and the edges shows the transition function. * The labeled on each edge is either a symbol in the set of  alphabet, ∑, or  denoting empty string. * This automation is nondeterministic because when it is in state-0 and the input symbol is a, it can either go to state-1 or stay in state-0. * The advantage of transition table is that it provides fast access to the transitions of states and the disadvantage is that it can take up a lot of soace. * In general, more than one sequence of moves can lead to an accepting state. If at least one such move ended up in a final state. For instance * The language defined by an NFA is the set of input strings that particular NFA accepts.   **Deterministic Finite Automata (DFA)**   * A deterministic finite automation is a special case of a non-deterministic finite automation (NFA) in which no state has an -transition for each state s and input symbol a, there is at most one edge labeled a leaving s.  * A DFA has st most one transition from each state on any input. It means that each entry on any input. It means that each entry in the transition table is a single state (as oppose to set of states in NFA). * Because of single transition attached to each state, it is vary to determine whether a DFA accepts a given input string. |
| **4** | **Describe the language for specifying lexical Analyzer.(16)**  ***LEX:***   * The main job of a lexical analyzer (scanner) is to break up an input stream into more usable elements (tokens)   a = b + c \* d;  ID ASSIGN ID PLUS ID MULT ID SEMI   * Lex is an utility to help you rapidly generate your scanners * Lexical analyzers tokenize input streams * Tokens are the terminals of a language   + English     - words, punctuation marks, …   + Programming language     - Identifiers, operators, keywords, … * Regular expressions define terminals/tokens * Lex source is a table of   + regular expressions and   + corresponding program fragments   ***digit [0-9]***  ***letter [a-zA-Z]***  ***%%***  ***{letter}({letter}|{digit})\* printf(“id: %s\n”, yytext);***  ***\n printf(“new line\n”);***  ***%%***  ***main() {***  ***yylex();***  ***}***   * Lex source is separated into three sections by %% delimiters * The general format of Lex source is * The absolute minimum Lex program is thus     ***{definitions}***  ***%%***  ***{transition rules}***  ***%%***  ***{user subroutines}***  **Lex Predefined Variables**   * yytext -- a string containing the lexeme * yyleng -- the length of the lexeme * yyin -- the input stream pointer   + the default input of default main() is stdin * yyout -- the output stream pointer   + the default output of default main() is stdout. * E.g.   ***[a-z]+ printf(“%s”, yytext);***  ***[a-z]+ ECHO;***  ***[a-zA-Z]+ {words++; chars += yyleng;}***   * yylex()   + The default main() contains a call of yylex() * yymore()   + return the next token * yyless(n)   + retain the first n characters in yytext * yywarp()   + is called whenever Lex reaches an end-of-file   The default yywarp() always returns 1 |
| **5** | **Prove that the following two regular expressions are equivalent by showing that minimum state DFA’s are same.**  (i)(a|b)\*#  (ii)(a\*|b\*)\*#  ***(Refer Class Notes)*** |
| **6** | **Draw the transition diagram to represent relational operators. (April/May,2017)**  **Give the REGULAR EXP FOR Unsigned integer is N+ (April/May,2017)**  **Draw the transition diagram for unsigned numbers.(6) Nov/Dec,2006,2007** |
| **7** | **Construct the NFA from the i.(a/b)\*a(a/b) & ii.(ab\*/ab) using Thompson‟s construction**  **algorithm.(10) May/June 2007 (Nov/Dec, 2017**) |
| **8** | **(i)Write notes on regular expression to NFA. Construct Regular expression to NFA for the sentence (a|b)\*a. (10) *(May 2016)***  **Thompson's construction is an NFA from a regular expression**.  The Thompson's construction is guided by the syntax of the regular expression with cases following the cases in the definition of regular expression.  ε is a regular expression that denotes {ε}, the set containing just the empty string. where i is a new start state and f is a new accepting state. This NFA recognizes {ε}.   1. If a is a symbol in the alphabet, a  ∑, then regular expression 'a' denotes {a} and the set containing just 'a' symbol. This NFA recognizes {a}.  1. Suppose, s and t are regular expressions denoting L{s} and L(t) respectively, then    1. s/r is a regular expression denoting L(s) L(t)    2. st is a regular expression denoting L(s) L(t) diagram    3. s\* is a regular expression denoting L(s)\* diagram    4. (s) is a regular expression denoting L(s) and can be used for putting parenthesis around regular expression   **(ii)Construct DFA to recognize the language (a/b)\*a.(6)** |
| **9** | **Write an algorithm for minimizing the number of states of a DFA.**  Minimization/optimization of a deterministic finite automata refers to detecting those states of a DFA whose presence or absence in a DFA does not affect the language accepted by the automata. Hence, these states can be eliminated from the automata without affecting the language accepted by the automata. Such states are:   * ***Unreachable States:*** Unreachable states of a DFA are not reachable from the initial state of DFA on any possible input sequence. * ***Dead States:*** A dead state is a nonfinal state of a DFA whose transitions on every input symbol terminates on itself. For example, *q* is a dead state if *q* is in *Q* *F*, and δ(*q*, *a*) = *q* for every *a* in Σ. * ***Nondistinguishable States:*** Nondistinguishable states are those states of a DFA for which there exist no distinguishing strings; hence, they cannot be distinguished from one another.   Therefore, optimization entails:   1. Detection of unreachable states and eliminating them from DFA; 2. Identification of nondistinguishable states, and merging them together; and 3. Detecting dead states and eliminating them from the DFA.   **Give the minimized DFA for the following expression. Nov/Dec,2006,2007 (Nov/Dec 2016)**  **(a/b)\*abb** |
| **10** | ***Write LEX specifications and necessary C code that reads English words from a text file and response every occurrence of the sub string ‘abc’ with ‘ABC’. The program should also compute number of characters, words and lines read. It should not consider and count any lines(s) that begin with a symbol ‘#’***  %{  #include <stdio.h>  #include <stdlib.h>  int cno = 0, wno = 0, lno = 0; /\*counts of characters, words and lines \*/  %}  character [a-z]  digit [0-9]  word ({character}|{digit})+[^({character}|{digit})]  line \n  %%  {line} { lno++; REJECT; }  {word} { wno++; REJECT; }  {character} { cno++; }  %%  void main()  { yylex();  fprintf(stderr, "Number of characters: %d; Number of words: %d; Number of lines: %d\n", cno, wno, lno);  return;  } |
| **11** | **Prove that the following two regular expressions are equivalent by showing that the minimum state DFA’s are same.**  **(i)(a/b)\* (ii)(a\*/b\*) (16) *(May 2015)***  (i)(a/b)  (ii) (a\*/b\*)  Input: DFA  Output: Minimized DFA  Step 1   * Draw a table for all pairs of states (Qi, Qj) not necessarily connected directly [All are unmarked initially]   Step 2   * Consider every state pair (Qi, Qj) in the DFA where Qi ∈ F and Qj ∉ F or vice versa and mark them. [Here F is the set of final states].   Step 3   * Repeat this step until we cannot mark anymore states − * If there is an unmarked pair (Qi, Qj), mark it if the pair {δ(Qi, A), δ (Qi, A)} is marked for some input alphabet.   Step 4  Combine all the unmarked pair (Qi, Qj) and make them a single state in the reduced DFA. |
| **12** | **Discuss how finite automata is used to represent tokens and perform lexical analysis with examples (Nov/Dec 2016)**  Lexical analysis is the process of reading the source text of a program and converting it into a sequence of tokens.  Since the lexical structure of more or less every programming language can be specified by a regular language, a common way to implement a lexical analyzer is to   1. Specify regular expressions for all of the kinds of tokens in the language. The disjunction of all of the regular expressions thus describes any possible token in the language. 2. Convert the overall regular expression specifying all possible tokens into a deterministic finite automaton (DFA). 3. Translate the DFA into a program that simulates the DFA. This program is the lexical analyzer.   This approach is so useful that programs called lexical analyzer generators exist to automate the entire process.  Two popular lexical analyzer generators are [flex](http://flex.sourceforge.net/) and [JFlex](http://jflex.de/).  If the lexical structure of the language is fairly simple, a hand-coded lexical analyzer can often be implemented easily. Usually, hand-coded lexical analyzers are implemented as a finite automaton, where a main program loop simulates the execution of the automaton and transitions between states. |
| **13** | **Differentiate between Tokens, Patterns and Lexemes (May/June,2016) (April/May,2017)**  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.  A **token** is a pair consisting of a token name and an optional attribute value. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes.  A **pattern** is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is more complex structure that is matched by many strings.   |  |  |  |  | | --- | --- | --- | --- | | **[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" | |   **(ii)What are the issues in lexical analysis?(4)**  There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing.  1. Simpler design is perhaps the most important consideration. The separation of lexical analysis from syntax analysis often allows us to simplify one or the other of these phases.  2. Compiler efficiency is improved.  3. Compiler portability is enhanced.  **(iii)Write notes on regular expressions. (6)**  Regular expression is used to define precisely the statements and expressions in the source language. For e.g. in Pascal the identifiers is denotes in the form of regular expression as letter **letter(letter|digit)\*.**  The algebraic law obeyed by regular expressions is called algebraic properties of regular expression. The algebraic properties are used to check equivalence of two regular expressions.   |  |  |  | | --- | --- | --- | | **S.No** | **Properties** | **Meaning** | | 1 | r1|r2=r2|r1 | | is commutative | | 2 | r1|(r1|r3)=(r1|r2)|r3 | | is associative | | 3 | (r1 r2)r3=r1(r2 r3) | Concatenation is associative | | 4 | r1(r2|r3)=r1 r2|r1 r3  (r2|r3) r1=r2 r1|r3 r1 | Concatenation is distributive over | | | 5 | Є r = r є = r | Є is identity | | 6 | r\*=(r|є)\* | Relation between є and \* | | 7 | r\*\*=r\* | \*is idempotent | |
| **14** | **Explain about input buffering.(8)**  **Input Buffering:**   * + Some efficiency issues concerned with the buffering of input.   + A two-buffer input scheme that is useful when lookahead on the input is necessary to identify tokens.   + Techniques for speeding up the lexical analyser, such as the use of sentinels to mark the buffer end.   + There are three general approaches to the implementation of a lexical analyser: * Use a lexical-analyser generator, such as Lex compiler to produce the lexical analyser from a regular expression based specification. In this, the generator provides routines for reading and buffering the input. * Write the lexical analyser in a conventional systems-programming language, using I/O facilities of that language to read the input. * Write the lexical analyser in assembly language and explicitly manage the reading of input.   ***Buffer pairs*:**   * + Because of a large amount of time can be consumed moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character. The scheme to be discussed:   + Consists a buffer divided into two N-character halves.      * N – Number of characters on one disk block, e.g., 1024 or 4096.   + Read N characters into each half of the buffer with one system read command.   + If fewer than N characters remain in the input, then eof is read into the buffer after the input characters.   + Two pointers to the input buffer are maintained.   + The string of characters between two pointers is the current lexeme.   + Initially both pointers point to the first character of the next lexeme to be found.   + Forward pointer, scans ahead until a match for a pattern is found.   + Once the next lexeme is determined, the forward pointer is set to the character at its right end.   + If the forward pointer is about to move past the halfway mark, the right half is filled with N new input characters.   + If the forward pointer is about to move past the right end of the buffer, the left half is filled with N new characters and the forward pointer wraps around to the beginning of the buffer.   –     ***Disadvantage of this scheme*:**   * + This scheme works well most of the time, but the amount of lookahead is limited.   + This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.   + For example: DECLARE ( ARG1, ARG2, … , ARGn ) in PL/1 program;   + Cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.   ***Sentinels*:**   * + In the previous scheme, must check each time the move forward pointer that have not moved off one half of the buffer. If it is done, then must reload the other half.   + Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.   + This can reduce the two tests to one if it is extend each buffer half to hold a sentinel character at the end.   + The sentinel is a special character that cannot be part of the source program. (eof character is used as sentinel).      * + In this, most of the time it performs only one test to see whether forward points to an eof.   + Only when it reach the end of the buffer half or eof, it performs more tests.   Since N input characters are encountered between eof’s, the average number of tests per input character is very close to 1. |
| **UNIT – III / PART – B C6660.3** | |
| **1** | **i. Find the language from (4)**  **S → 0S1 | 0A1 A→1A0 | 10**  ***S→0S1 | 0A | 0 |1B | 1***  ***A→0A | 0***  ***B→1B|1***  **Answer:**  The minimum string is S-> 0 | 1  S->0S1=>001  S->0S1=>011  S->0S1=>00S11=>000S111=>0000A111=>00000111  Thus L={ 0n 1 m | m not equal to n, and n,m >=1}  **ii.Define Parse tree , Regular Expression , Left most derivation , Right most derivation, and write example for each.** (4)  A parse tree or parsing tree or derivation tree or (concrete) syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.    **Regular Expression**  Regular expressions are mathematical symbolism which describes the set of strings of specific language. It provides convenient and useful notation for representing tokens. Here are some rules that describe definition of the regular expressions over the input set denoted by ∑  **Leftmost Derivation**  A top-down parse we always choose the leftmost non-terminal in a sentential form to apply a production rule to - this is called a leftmost derivation.    **Rightmost Derivation**  A bottom-up parse then the situation would be reversed, and we would want to do apply the production rules in reverse to the leftmost symbols; thus we are performing a rightmost derivation in reverse.    **iii.Write algorithm to convert NFA from Regular expression. (4)**  We will use the rules which defined a regular expression as a basis for the construction:   * The NFA representing the empty string * If the regular expression is just a character, eg. a, then the corresponding NFA * The union operator is represented by a choice of transitions from a node * Concatenation simply involves connecting one NFA to the other   The Kleene closure must allow for taking zero or more instances of the letter from the input |
| **2** | **Explain context free grammar with examples**  ***Definition*** − A context-free grammar (CFG) consisting of a finite set of grammar rules is a quadruple **(N, T, P, S)** where   * **N** is a set of non-terminal symbols. * **T** is a set of terminals where **N ∩ T = NULL.** * **P** is a set of rules, **P: N → (N ∪ T)\***, i.e., the left-hand side of the production rule **P** does have any right context or left context. * **S** is the start symbol.   **Example**   * The grammar ({A}, {a, b, c}, P, A), P : A → aA, A → abc. * The grammar ({S, a, b}, {a, b}, P, S), P: S → aSa, S → bSb, S → ε * The grammar ({S, F}, {0, 1}, P, S), P: S → 00S | 11F, F → 00F | ε  Generation of Derivation Tree A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar. Representation Technique  * **Root vertex** − Must be labeled by the start symbol. * **Vertex** − Labeled by a non-terminal symbol. * **Leaves** − Labeled by a terminal symbol or ε.   If S → x1x2 …… xn is a production rule in a CFG, then the parse tree / derivation tree will be as follows −  There are two different approaches to draw a derivation tree −  **Top-down Approach −**   * Starts with the starting symbol **S** * Goes down to tree leaves using productions   **Bottom-up Approach −**   * Starts from tree leaves * Proceeds upward to the root which is the starting symbol **S**  Derivation or Yield of a Tree The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.  **Example**  Let a CFG {N,T,P,S} be  N = {S}, T = {a, b}, Starting symbol = S, P = S → SS | aSb | ε  One derivation from the above CFG is “abaabb”  S → SS → aSbS → abS → abaSb → abaaSbb → abaabb |
| **3** | **i. Prove the grammar is ambiguous. (4)**  **E→E+E | E\*E | (E) | id**    **ii. Specify the demerits of ambiguous grammar. (2)**  A grammar is ambiguous if, for any string  􀂄 it has more than one parse tree, or  􀂄 there is more than one right-most derivation, or  􀂄 there is more than one left-most derivation  (the three conditions are equivalent)  **iii. What are the rules to convert an unambiguous grammar from ambiguous grammar**  Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.  iv. **Using unambiguous grammar, write Leftmost derivation ,draw parse tree for the string**  **id\*id\*id+id\*id**  E 🡪 E \* E  E 🡪 E \* E \* E  E 🡪 id \* E \* E  E 🡪 id \* id \* E  E 🡪 id \* id \* E + E  E 🡪 id \* id \* id + E  E 🡪 id \* id \* id + E \* E  E 🡪 id \* id \* id + id \* E  E 🡪 id \* id \* id + id \* id |
| **4** | **(i) Construct stack implementation of shift reduce parsing for the grammar (May 2016)**  **E-> E+E**  **E -> E\*E**  **E -> (E)**  **E -> id and the input string id1+id2\*id3. (8)**  **Solution:**  **Describe the conflicts that may occur during shift reduce parsing**   * Shift/Reduce conflict: The entire stack contents and the next input symbol cannot decide whether to shift or reduce. * Reduce/Reduce conflict: The entire stack contents and the next input symbol cannot decide which of several reductions to make. |
| **5** | **Construct a recursive decent parser for the following grammar and write the algorithm for elimination of left recursion and give the stack implementation for the sentence “ id+id\*id $”**  **E→E+T | T**  **T→T\*F | F**  **F→(E) | id**  **Algorithm to eliminate left recursion:**  1. Arrange the non-terminals in some order A1, A2 . . . An.  2. **for** *i*:= 1 **to** *n* **do begin**  **for** *j*:= 1 **to** *i*-1 **do begin**  replace each production of the form Ai → A j γ by the  productions Ai → δ1 γ | δ2γ | . . . | δk γ  where Aj → δ1 | δ2 | . . . | δk are all the current Aj-productions;  **end**  eliminate the immediate left recursion among the Ai-productions  **end**  **Solution:**  After eliminating the left-recursion the grammar becomes,  E → TE‟  E‟ → +TE‟ | ε  T → FT‟  T‟ → \*FT‟ | ε  F → (E) |id |
| **6** | **Write in detail about Recursive Predictive parser and Non-Recursive Predictive parser (Nov/Dec, 2017)**  Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.  It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.  But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.  This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.  **Back-tracking**  Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:  S → rXd | rZd  X → oa | ea  Z → ai  For an input string: read, a top-down parser, will behave like this:  It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).  Now the parser matches all the input letters in an ordered manner. The string is accepted.   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  |   **Non-Recursive Predictive parser:**  It is possible to build a non-recursive predictive parser. This is done by maintaining an explicit stack and using a table. Such a parser is called a table-driven parser. The non-recursive LL(1) parser looks up the production to apply by looking up a parsing table. The LL(1) table has one dimension for current non-terminal to expand and another dimension for next token. Each table cell contains one production.  The input buffer contains the string to be parsed; $ is the end-of-input marker. The stack contains a sequence of grammar symbols. Initially, the stack contains the start symbol of the grammar on the top of $. The parser is controlled by a program that behaves as follows:  1. The program considers X, the symbol on top of the stack, and a, the current input symbol. These two symbols, X and a determine the action of the parser. There are three possibilities.  X = a = $, the parser halts and announces successful completion  2.X = a ≠ $ the parser pops X off the stack and advances input pointer to next input symbol**.**  3.If X is a non-terminal, the program consults entry M[X,a] of parsing table M.  a.If the entry is a production M[X,a] = {X 🡪 UVW }, the parser replaces X on top of the stack by WVU (with U on top). As output, the parser just prints the production used: X 🡪 UVW. However, any other code could be executed here.  b.If M[X,a] =error, the parser calls an error recovery routine |
| **7** | **Explain LR parsing algorithm. (Nov/Dec, 2017)**  LR parser consists of an input, an output, a stack, a driver program and a parsing table that has two functions  1. Action  2. Goto  The driver program is same for all LR parsers. Only the parsing table changes from one parser to another.  The parsing program reads character from an input buffer one at a time, where a shift reduces parser would shift a symbol; an LR parser shifts a state. Each state summarizes the information contained in the stack.  The stack holds a sequence of states, *so,*s1, · ·· , *Sm,*where *Sm*is on the top.    **Action**This function takes as arguments a state *i*and a terminal *a*(or $, the input end marker). The value of ACTION [i, *a]*can have one of the four forms:  i) Shift *j,*where *j*is a state.  ii) Reduce by a grammar production A---> *β.*  iii) Accept.  iv) Error.  **Goto**This function takes a state and grammar symbol as arguments and produces a state.  If GOTO [Ii ,A] = Ij, the GOTO also maps a state *i*and non terminal A to state *j.*  **Behavior of the LR parser**  1. If ACTION[sm, *ai]*= shift *s.*The parser executes the shift move, it shifts the next state *s*onto the stack, entering the configuration  a) *Sm -*the state on top of the stack.  b) *ai-*the current input symbol.  2. If ACTION[sm, *ai]*=reduce A---> β*,*then the parser executes a reduce move, entering the configuration  *(s0s1 ... S(m-r)S,*ai+l ... *an$)*  a) where *r*is the length of *β*and s= *GOTO[sm - r,*A].  b) First popped *r*state symbols off the stack, exposing state *Sm-r·*  c) Then pushed *s,*the entry for *GOTO[sm-r,*A], onto the stack.  3. If ACTION[sm, *ai]*= accept, parsing is completed.  4. If ACTION[sm, *ai]*= error, the parser has discovered an error and calls an error recovery routine. **LR Parsing Algorithm** **Algorithm**LR Parsing Algorithm.  **Input**   Input string w,         LR-Parsing table with functions ACTION and         GOTO for a grammar G  Output If w is in L(G), the reduction steps of a         bottom-up parse for w,         otherwise, an error indication.  Method Initially, the parser has So on its stack,         where So is the initial state, and w $ in the         input buffer.         let a be the first symbol of w $         while(l) { //repeat forever         let s be the state on top of the stack;         if(ACTION[s, a] =shift t {         push t onto the stack;         let a be the next input symbol;         } else if (ACTION [s, a] = reduce A---> *β*) {         pop *β* symbols off the stack;         let state t now be on top of the stack;         push GOTO[t, A] onto the stack;         output the production A---> *β;*         } else if (ACTION [s, a] accept) break;         //parsing is done         else call error-recovery routine;                 } **LR(O) Items** An LR(O) item of a grammar G is a production of G with a dot at some position of the body.  (eg.)                                                     A ---> •XYZ                                                     A ---> XeYZ                                                     A ---> XYeZ                                                     A ---> XYZ•  One collection of set of LR(O) items, called the canonical LR(O) collection, provides finite automaton that is used to make parsing decisions. Such an automaton is called an LR(O) automaton. |
| **8** | **Construct Predictive Parser (Non-Recursive Predictive parser ) for the following grammar and write the algorithm for FIRST AND FOLLOW and find moves made by predictive parser on input id+id\*id**  **E→E+T | T**  **T→T\*F | F**  **F→(E) | id (May/June-2012&13) (Nov/Dec 2016) (Nov/Dec, 2017)**  **Rules for first( ):**   1. If *X* is terminal,  [then FIRST(*X*) is {X}](http://notes.pmr-insignia.org/). 2. If *X* → ε is  [a production, then add ε to FIRST(*X*)](http://notes.pmr-insignia.org/). 3. If *X* is non- [terminal and X → *a*α is a production then add *a* to FIRST(X)](http://notes.pmr-insignia.org/). 4. If X is non- [terminal and *X* → *Y1* *Y2*…*Yk* is a production, then place *a* in FIRST](http://notes.pmr-insignia.org/)(*X*) if for some *i*, *a* is in  [FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Y](http://notes.pmr-insignia.org/)i-*1* => ε. If ε isin FIRST(*Y* [*j*) for all j=1,2,..,k, then add ε to FIRST(*X*)](http://notes.pmr-insignia.org/).   **Rules for follow(**  [**)**](http://notes.pmr-insignia.org/)**:**   1. If *S* is a start  [symbol, then FOLLOW(*S*) contains $](http://notes.pmr-insignia.org/). 2. If there is  [a production *A* → α*B*β, then everything in FIRST(β) except ε](http://notes.pmr-insignia.org/) is placed in follow(*B*). 3. If there is  [a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β)](http://notes.pmr-insignia.org/) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).   After eliminating left-recursion the grammar is  E → TE’  E’ → +TE’ |ε  T → FT’  T’ → \*FT’ | ε  F → (E) |id  **First( ) :**  FIRST(E) ={  [(, id](http://notes.pmr-insignia.org/)}  FIRST(E’) ={+  [, ε](http://notes.pmr-insignia.org/) }  FIRST(T) = {  [( , id](http://notes.pmr-insignia.org/)}  FIRST(T’) ={\*,  [ε](http://notes.pmr-insignia.org/) }  FIRST(F) ={  [( , id](http://notes.pmr-insignia.org/) }  **Follow( ):**  FOLLOW(E)  [={ $, )](http://notes.pmr-insignia.org/) }  FOLLOW(E’)  [={ $, )](http://notes.pmr-insignia.org/) }  FOLLOW(T)  [={ +, $, )](http://notes.pmr-insignia.org/) }  FOLLOW(T’) = { +, $, ) }  FOLLOW(F) ={+, \* , $ , ) }   |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **Predictive parsing table :** | | |  |  |  |  |  |  |  |  |  |  |  |  | |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  | NON- | id | + | \* | ( | ) | $ |  |  | |  |  |  |  |  |  |  |  |  | |  | TERMINAL |  |  |  |  |  |  |  |  |  |  |  |  |  |  | |  | E |  | E → TE’ |  |  |  |  |  | E → TE’ |  |  |  |  |  |  | |  | E’ |  |  |  | E’ → +TE’ |  |  |  |  |  | E’ → ε |  | E’→ ε |  |  | |  | T |  | T → FT’ |  |  |  |  |  | T → FT’ |  |  |  |  |  |  | |  | T’ |  |  |  | T’→ ε |  | T’→ \*FT’ |  |  |  | T’ → ε |  | T’ → ε |  |  | |  | F |  | F→ id |  |  |  |  |  | F→ (E) |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | |  | **Stack implementation:** | | | | | |  |  | |  |  |  |  |  |  |  |  |  | |  | **stack** |  | **Input** | **Output** |  |  | |  |  |  |  |  |  | |  | $E |  |  | id+id\*id $ |  |  |  |  | |  | $E’T |  |  | id+id\*id $ |  | E → TE’ |  |  | |  | $E’T’F |  |  | id+id\*id $ |  | T → FT’ |  |  | |  | $E’T’id |  |  | id+id\*id $ |  | F→ id |  |  | |  | $E’T’ |  |  | +id\*id $ |  |  |  |  | |  | $E’ |  |  | +id\*id $ |  | T’ → ε |  |  | |  | $E’T+ |  |  | +id\*id $ |  | E’ → +TE’ |  |  | |  | $E’T |  |  | id\*id $ |  |  |  |  | |  | $E’T’F |  |  | id\*id $ |  | T → FT’ |  |  | |  | $E’T’id |  |  | id\*d $ |  | F→ id |  |  | |  | $E’T’ |  |  | \*id $ |  |  |  |  | |  | $E’T’F\* |  |  | [\*id](http://notes.pmr-insignia.org/) $ |  | [T’ → \*FT](http://notes.pmr-insignia.org/)’ |  |  | |  | $E’T’F |  |  | [id](http://notes.pmr-insignia.org/) $ |  |  |  |  | |  | $E’T’id |  |  | [id](http://notes.pmr-insignia.org/) $ |  | [F→ i](http://notes.pmr-insignia.org/)d |  |  | |  | $E’T’ |  |  | $ |  |  |  |  | |  | $E’ |  |  | $ |  | [T’ →](http://notes.pmr-insignia.org/) ε |  |  | |  | $ |  |  | $ |  | [E’ →](http://notes.pmr-insignia.org/) ε |  |  | |
| **9** | **Check whether the following grammar is a LL(1) grammar**  **S 🡪 iEtS | iEtSeS | a**  **E 🡪 b**  **Also define the FIRST and FOLLOW procedures (May/June,2016)**   1. If *X* is terminal,  [then FIRST(*X*) is {X}](http://notes.pmr-insignia.org/). 2. If *X* → ε is  [a production, then add ε to FIRST(*X*)](http://notes.pmr-insignia.org/). 3. If *X* is non- [terminal and X → *a*α is a production then add *a* to FIRST(X)](http://notes.pmr-insignia.org/). 4. If X is non- [terminal and *X* → *Y1* *Y2*…*Yk* is a production, then place *a* in FIRST](http://notes.pmr-insignia.org/)(*X*) if for some *i*, *a* is in  [FIRST(Yi), and ε is in all of FIRST(Y1),…,FIRST(Yi-1); that is, Y1,….Y](http://notes.pmr-insignia.org/)i-*1* => ε. If ε isin FIRST(*Y* [*j*) for all j=1,2,..,k, then add ε to FIRST(*X*)](http://notes.pmr-insignia.org/).   **Rules for follow(**  [**)**](http://notes.pmr-insignia.org/)**:**   1. If *S* is a start  [symbol, then FOLLOW(*S*) contains $](http://notes.pmr-insignia.org/). 2. If there is  [a production *A* → α*B*β, then everything in FIRST(β) except ε](http://notes.pmr-insignia.org/) is placed in follow(*B*). 3. If there is  [a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β)](http://notes.pmr-insignia.org/) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).   After eliminating left factoring, we have  S → iEtSS‟ |a  S‟→ eS | ε  E → b  To construct a parsing table, we need FIRST()and FOLLOW() for all the non-terminals.  FIRST(S) ={ i, a }  FIRST(S‟) = {e, ε }  FIRST(E) ={ b}  FOLLOW(S) ={ $ ,e }  FOLLOW(S‟) = { $ ,e }  FOLLOW(E) = {t}  **Parsing table:**  Since there are more than one production, the grammar is not LL(1) grammar.  **Actions performed in predictive parsing:**  1. Shift  2. Reduce  3. Accept  4. Error  **Implementation of predictive parser:**  1. Elimination of left recursion, left factoring and ambiguous grammar.  2. Construct FIRST() and FOLLOW() for all non-terminals.  3. Construct predictive parsing table.  4. Parse the given input string using stack and parsing table. |
| **10** | **Construct non recursion predictive parsing table for the following grammar. And write the**  **algorithm,**  **E 🡪E or E /E and E/ not E / (E) / 0 /1. (Dec-12,Marks 16). (May/June,2016)**  **(or)**  **E-->E or T/T**  **T-->T and F/F**  **F-->not G**  **G-->(E) /0/1**  **(or)**  **bexpr→bexpr OR bterm | bterm**  **bterm→bterm AND bfactor | bfactor**  **bfactor→NOT bfactor | (bexpr) | true | false**  **Refer Class Note**   * Find FIRST and FOLLOW and * construct table and parse the string. |
| **11** | **Consider the following grammar (Nov/Dec-2012) (Nov/Dec 2016) ( April/May,2017)**  **E→E+T | T, T→TF | F, F→F\* | a | b construct the SLR parsing table for this grammar. Also parse the input a\*b+a .** |
| **12** |  |
| **13** | **(i)Construct SLR parsing table for the following grammar (10). (Nov/Dec-2012,April/May-04)**  **S 🡪 L=R|R L 🡪\*R| id R 🡪 L** |
| **14** | **Parse the string (a,a) using SLR parsing table.**  **S→ (L) | a**  **L→L , S | S**  After left recursion elimination, it becomes  0) G := S $  1) S ::= ( L )  2) S ::= a  3) L ::= S L'  4) L' ::= , S L'  5) L' ::=  The first/follow tables are:  which are used to create the parsing table: |
| **15** | **Construct a predictive Parsing table for the grammar. ( April/May,2017)**  **S→ (L) | a**  **L→L , S | S**  **And show whether (a,(a,(a,a))) is accepted or not.**  After left recursion elimination, it becomes  0) G := S $  1) S ::= ( L )  2) S ::= a  3) L ::= S L'  4) L' ::= , S L'  5) L' ::=  The first/follow tables are:  which are used to create the parsing table: |
| **16** | **Generate SLR Parsing table for the following grammar.**  **S-->Aa|bAc|Bc|bBa**  **A-->d**  **B-->d**  **And parse the sentences “bdc”and “dd”.(April/May 2015)**  1. Construct F = {I0, I1, ... In}, the collection of LR(0) configurating sets for G'.  2. State i is determined from Ii. The parsing actions for the state are determined as  follows:  a) If A –> u• is in Ii then set Action[i,a] to reduce A –> u for all a in Follow(A) (A is not S').  b) If S' –> S• is in Ii then set Action[i,$] to accept.  c) If A –> u•av is in Ii and successor(Ii, a) = Ij, then set Action[i,a] to shift j (a must be a terminal).  3. The goto transitions for state i are constructed for all non­terminals A using the  rule:  If successor(Ii, A) = Ij, then Goto [i, A] = j.  4. All entries state is the one constructed from the configurating set containing  S' –> •S not defined by rules 2 and 3 are errors. |
| **17** | **Construct CLR parsing table to parse the sentence id=id\*id for the following grammar.**  **S→ L=R | R**  **L→\*R | id**  **R→L**  **Solution:**   * Find LR(1) items; * Construct CLR table. * Parse the string. |
| **18** | **Construct LALR parsing table for the grammar.**  **E→ E+T | T**  **T→ T\*F | F**  **F→ (E) | id**  **Solution:**   * Find LR(1) items. * Find same core items different second component then merge it * After merging construct LALR parsing table. |
| **19** | **Write in detail about**   1. **Recursive descent parsing with algorithm.** 2. **Top down parsing**  Recursive Descent Parsing Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity.  This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.  But the grammar associated with it (if not left factored) cannot avoid back-tracking. Back-tracking Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:  S → rXd | rZd  X → oa | ea  Z → ai  For an input string: read, a top-down parser, will behave like this:  It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).  Now the parser matches all the input letters in an ordered manner. The string is accepted.   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  |   **TOP-DOWN PARSING**  It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.  **Types of top-down parsing :**  1. Recursive descent parsing  2. Predictive parsing  **1. RECURSIVE DESCENT PARSING**  Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.**  This parsing method may involve **backtracking**, that is, making repeated scans of the input.  **Example for backtracking :**  Consider the grammar G : S → cAd  A → ab |a  and the input string w=cad.  The parse tree can be constructed using the followingtop-down approach :  **Step1:**  Initially create a tree with single node labeled S. An input pointer points to „c‟, the first symbol  of w. Expand the tree with the production of S.  **Step2:**  The leftmost leaf „c‟ matches the first symbol of w, so advance the input pointer to the second symbol of w „a‟ and consider the next leaf „A‟. Expand A using the first alternative.  **Step3:**  The second symbol „a‟ of w also matches with second leaf of tree. So advance the input pointer to third symbol of w „d‟**.** But the third leaf of tree is b which does not match with the input symbol **d.** Hence discard the chosen production and reset the pointer to second position. This is called **backtracking.**  **Step4:**  Now try the second alternative for A.  Now we can halt and announce the successful completion of parsing. |
| **20** | **(i) Write the algorithm to eliminate left-recursion and left-factoring and apply both to the following grammar. (8)**  **E-->E+T|E-T|T**  **T-->a|b|(E) ( April/May 2015)**  Step 1: Eliminate left recursion and left factoring  Step 2: Find out the FIRST  Step 3: Find out FOLLOW  Step 4: Constructing a parsing table  Step 5: Stack Implementation |
| **UNIT – IV / PART –B CS6660.4** | |
|  | **Explain the concept of syntax directed definition.**   * A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions. * The semantic rules define values for attributes associated with the symbols of the productions. * These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass. * SDDs are useful for specifying translations.   CFG + semantic rules = Syntax Directed Definitions   * A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. * If *X* is a symbol and *a* is one of its attributes, then we write *X.a* to denote the value of a at a particular parse-tree node labeled *X.* * If we implement the nodes of the parse tree by records or objects, then the attributes of *X* can be implemented by data fields in the records that represent the nodes for *X.* * Attributes may be of any kind: numbers, types, table references, or strings, for instance.   The strings may even be long sequences of code, say code in the intermediate language used by a compiler. |
|  | **Construct parse tree, syntax tree and annotated parse tree for the input string is 5\*6+7;**  **Parse tree**    **Syntax tree**    **Annotated parse tree** |
|  | **Explain 1)Synthesized attribute 2)inherited attribute with suitable examples.**  **Synthesized attributes**   * A *synthesized attribute* for a nonterminal *A* at a parse-tree node *N* is defined by a semantic rule associated with the production at *N.* * Note that the production must have *A* as its head. * A synthesized attribute at node *N* is defined only in terms of attribute values at the children of *N* and at *N* itself. * These attributes get values from the attribute values of their child nodes.   ***Example***  1. 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.  2. E → E + T  The parent node E gets its value from its child node E and T.  ***Example***  E → E1 + T { E.val = E1.val + T.val; }    E → T { E.val = T.val; }    T → ( E ) { T.val = E.val; }    T → digit { T.val = digit.lexval; }  **Inherited attributes**   * An *inherited attribute* for a nonterminal B at a parse-tree node *N* is defined by a semantic rule associated with the production at the parent of *N.* * Note that the production must have *B* as a symbol in its body. * An inherited attribute at node *N* is defined only in terms of attribute values at JV's parent, *N* itself, and *N's* siblings. * Inherited attributes can take values from parent and/or siblings. * Example   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 A { E.node = A.s;  A.i = T.node; }    A → + T A1 { A1.i = Node('+', A.i, T.node);  A.s = A1.s; }    A → e { A.s = A.i; }    T → ( E ) { T.node = E.node; }    T → id { T.node = Leaf(id, id.entry); } |
|  | **Write a syntax directed definition and evaluate 9\*3+2 with parser stack using LR parsing method.**   * Parse tree helps us to visualize the translation specified by SDD. * The rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. * A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.   With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree. |
|  | **Consider the following CFG,**  **E→TR**  **R→+TR**  **R→-TR**  **R→є**  **T→num**  With translation scheme to generate to generate postfix expression equivalent to the given infix expression which is recognized by above grammar. All actions in the translation should be at the end of each production  With translation scheme to generate to generate postfix expression equivalent to the given infix expression which is recognized by above grammar. All actions in the translation should be at the end of each production |
|  | Explain the  Implementing L-Attributed SDD's. The second class of SDD's is called *L-attributed definitions.* The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence "L-attributed").  More precisely, each attribute must be either  1. Synthesized, or  2. Inherited, but with the rules limited as follows.  Suppose that there is a production *A* ->• *XXX2 • • • Xn,* and that there is an inherited attribute ***Xi.a***computed by a rule associated with this production.  Then the rule may use only:  (a) Inherited attributes associated with the head *A.*  (b) Either inherited or synthesized attributes associated with the occurrences of symbols *X±,X2,... , X^* located to the left of *X{.*  (c) Inherited or synthesized attributes associated with this occurrence of *Xi* itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this Xj. |
|  | **(i)Given the Syntax-Directed Definition below construct the annotated parse tree for the input expression: “int a, b, c”.**  D → T L L.inh = T.type  T → int T.type = integer  T → float T.type = float  L → L1, id L1.inh = L.inh addType(id.entry,L.inh)  L → id addType(id.entry,L.inh)  (ii) Given the Syntax-Directed Definition below with the synthesized attribute val, draw the annotated parse tree for the expression (3+4) \* (5+6).  L → E L.val = E.val  E → T E.val = T.val  E → E1 + T E.val = E1.val + T.val  T → F T.val = F.val  T → T1 \* F T.val = T1.val \* F.val  F → ( E ) F.val = E.val  F → digit F.val = digit.lexval  D → T L L.inh = T.type  T → intT.type = integer  T → float T.type = float  L → L1, id L1.inh = L.inhaddType(id.entry,L.inh)  L → id addType(id.entry,L.inh)    **(ii) Given the Syntax-Directed Definition below with the synthesized attribute val, draw the annotated parse tree for the expression (3+4) \* (5+6).**  L → E L.val = E.val  E → T E.val = T.val  E → E1 + T E.val = E1.val + T.val  T → F T.val = F.val  T → T1 \* F T.val = T1.val \* F.val  F → ( E ) F.val = E.val  F → digit F.val = digit.lexval |
|  | **Explain the various structures that are used for the symbol table constructions.(April/may 2012,2014)**   * A separate array ‘arr\_lexemes’ holds the character string forming an identifier. The string is terminated by an end-of-string character, denoted by EOS, that may not appear in identifiers. * Each entry in symbol-table array ‘arr\_symbol\_table’ is a record consisting of two fields, as “lexeme\_pointer”, pointing to the beginning of a lexeme, and token. * Additional fields can hold attribute values. 0th entry is left empty, because lookup return 0 to indicate that there is no entry for a string. * The 1st, 2nd, 3rd, 4th, 5th, 6th, and 7th entries are for the ‘a’, ‘plus’ ‘b’ ‘and’, ‘c’, ‘minus’, and ‘d’ where 2nd, 4th and 6th entries are for reserve keyword.   **List**    **Self Organizing List**    **Hash Table**    **Search Tree** |
|  | **Specify a type checker which can handle expressions, statements and functions. (Nov/Dec, 2017)**  There are three different storage allocation strategies based on this division of run-time storage. The strategies are-  1. **Static allocation**- The static allocation is for all the data objects at compile time.  2. **Stack allocation**- In the stack allocation a stack is used to manage the run time storage.  3. **Heap allocation**- In heap allocation the heap is used to manage the dynamic memory allocation.  **Static Allocation**  The size of data objects is known at compile time. The names of these objects are bound to storage at compile time only and such an allocation of data objects is done by static allocation.  The binding of name with the amount of storage allocated do not change at run-time. Hence the name of this allocation is static allocation.  In static allocation the compiler can determine the amount of storage required by each data object. And therefore it becomes easy for a compiler to find the addresses of these data in the activation records.  At compile time compiler can fill the addresses at which the target code find the data it operates on.  FORTRAN uses the static allocation strategy.  **Stack Allocation**  Stack allocation strategy is a strategy in which the storage is organized as stack. This stack is also called control stack.  As activation begins the activation records are pushed onto the stack and on completion of this activation the corresponding activation records can be popped.  The locals are stored in the each activation record. Hence locals are bound to corresponding activation record on each fresh activation.  The data structures can be created dynamically for stack allocation.  **Heap Allocation**  If the values of non-local variables must be retained even after the activation record then such a retaining is not possible by stack allocation. This limitation of stack allocation is because of its Last In First Out nature. For retaining of such local variables heap allocation strategy is used.  The heap allocation allocates the continuous block of memory when required for storage of activation records or other data object. This allocated memory can be deallocated when activation ends. This deallocated space can be further reused by heap manager.  The efficient heap management can be done by,  i) Creating a linked list for the free blocks and when any memory is deallocated that block of memory is appended in the linked list.  ii) Allocate the most suitable block of memory from the linked list. i.e., use best fit technique for allocation of block.  ***(ii)Specify a type checker which can handle expressions, statements and functions.(8)*** The validity of statements Expressions have types, but statements do not.  However, also statements are checked in type checking.  We need a new judgement form, saying that a statement S is valid:  F,G => S valid  Example: typing rule for an assignment  F,G => e : T  -------------------- x : T is in G  F,G => x = e ; valid  Example: typing rule for while loops  F,G => e : bool F,G => S valid  --------------------------------  F,G => while (e) S valid  **Expression**  We prove that int x ; x = x + 5 ; is valid in the empty context ().  x : int => x : int x : int => 5 : int  -----------------------------------------  x : int => x + 5 : int  --------------------------  x : int => x = x + 5 ; valid  -------------------------------  () => int x ; x = x + 5 ; valid  The signature is omitted for simplicity. Function types No expression in the language has **function types**, because functions are never returned as values or used as arguments.  However, the compiler needs internally a data structure for function types, to hold the types of the parameters and the return type. E.g. for a function  bool between (int x, double a, double b) {...}  we write  between : (int, double, double) -> bool  to express this internal representation in typing rules. |
|  | **Explain the organization of runtime storage in detail. (Nov/Dec, 2017)**  **STATIC STORAGE ALLOCATION**  In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:  1. The size of each object must be known at compile time.  2. Only one occurrence of each object is allowable at a given moment during program execution.  A static storage-allocation strategy is very simple to implement.  An object address can be either an absolute or a relative address.  **DYNAMIC STORAGE ALLOCATION**  In a dynamic storage-allocation strategy, the data area requirements for a program are not known entirely at compilation time. In particular, the two criteria that were given in the previous section as necessary for static storage allocation do not apply for a dynamic storage-allocation scheme. The size and number of each object need not be known at compile time; however, they must be known at run time when a block is entered. Similarly more than one occurrence of a data object is allowed, provided that each new occurrence is initiated at run time when a block is entered. |
|  | **What are different storage allocation strategies? Explain. (May/June,2016) (April/May,2017)**  In a static storage-allocation strategy, it is necessary to be able to decide at compile time exactly where each data object will reside at run time. In order to make such a decision, at least two criteria must be met:  1. The size of each object must be known at compile time.  2. Only one occurrence of each object is allowable at a given moment during program execution.  A static storage-allocation strategy is very simple to implement.  An object address can be either an absolute or a relative address |
|  | **Explain any 4 issues in storage allocation (4). (April/May 2015)**   * The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. * These issues are tricky because the same name in a program text can refer to multiple locations at run time. * The two adjectives *static* and *dynamic* distinguish between compile time and run time, respectively. * We say that a storage-allocation decision is static, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes.   Conversely, a decision is *dynamic* if it can be decided only while the program is running. |
|  | **Give a Syntax directed Definitions to differentiate expressions formed by applying the arithmetic operators + and \* to the variable X and constants ; expression :X\*(3\*X+X\*X). ( April/May 2015) (8)**   * A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions. * The semantic rules define values for attributes associated with the symbols of the productions. * These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass. * SDDs are useful for specifying translations.   CFG + semantic rules = Syntax Directed Definitions  Example  E → E1 + T { E.val = E1.val + T.val; }    E → T { E.val = T.val; }    T → ( E ) { T.val = E.val; }    T → digit { T.val = digit.lexval; } |
|  | **For the given program fragment A[i,j]=B[i,k] do the following:**  **(i)Draw the annotated parse tree with the translation scheme to convert to three address code (6)**  **(ii) Write the 3-address code(6)**  **(iii)Determine the address of A[3,5] where , all are integer arrays with size of A as 10\*10 and B as 10\*10 with k=2 and the start index position of all arrays is at 1.(assume the base addresses) (4) (April/May 2015)**  **Annotated parse Tree:**    (ii) Write the 3-address code(6)  **Translation scheme**    In three-address code, this would be broken down into several separate instructions. These instructions translate more easily to assembly language. It is also easier to detect common sub-expressions for shortening the code.  **Example:**  t1 := b \* b  t2 := 4 \* a  t3 := t2 \* c  t4 := t1 - t3  t5 := sqrt(t4)  t6 := 0 - b  t7 := t5 + t6  t8 := 2 \* a  t9 := t7 / t8  x := t9  **(iii)Determine the address of A[3,5] where , all are integer arrays with size of A as 10\*10 and B as 10\*10 with k=2 and the start index position of all arrays is at 1.(assume the base addresses) (4) (May 2015)**   * Array indexing- In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value. * Base address consists of the address of first element in an array. * Other elements of the array can be accessed using the base address and offset value.   ***Example: x = y[i]***  ***Memory location m = Base address of y + Displacement i***   * x = contents of memory location m similarly x[i] = y * Memory location m = Base address of x + Displacement i.   The value of y is stored in memory location m |
|  | **(i).Apply Back-patching to generate intermediate code for the following input.**  **x:2+y;**  **If x<y then x:=x+y;**  **repeat y:=y\*2;**  **while x>10 do x:=x/2;**  **Write the semantic rule and derive the Parse tree for the given code (12)**  **(ii) What is an Activation Record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations. (4) (April/May 2015)**  A key problem when generating code for boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump. For example, the translation of the boolean expression B in if (B) S contains a jump, for when B is false, to the instruction following the code for S. In a one-pass translation, B must be translated before S is examined. What then is the target of the goto that jumps over the code for S?this problem is addressed by passing labels as inherited attributes to where the relevant jump instructions were generated. But a separate pass is then needed to bind labels to addresses. This section takes a complementary approach, called backpatching, in which lists of jumps are passed as synthesized attributes. Specifically, when a jumpis generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined. All of the jumps on a list have the same target label.  **(ii)What is an Activation Record? Explain how its relevant to the intermediate code generation phase with respect to procedure declarations. (4) (April/May 2015)**  Modern imperative programming languages typically have local variables.  – Created upon entry to function.  – Destroyed when function returns.  Each invocation of a function has its own instantiation of local variables.  – Recursive calls to a function require several instantiations to exist simultaneously.  – Functions return only after all functions it calls have returned last-in-first-out  (LIFO) behavior.  – A LIFO structure called a stack is used to hold each instantiation.  The portion of the stack used for an invocation of a function is called the function’s  stack frame or activation record.ch7-11-728  **The Stack**   * Used to hold local variables. * Large array which typically grows downwards in memory toward lower addresses, * shrinks upwards. * Push(r1):   stack\_pointer--;  M[stack\_pointer] = r1;   * r1 = Pop():   r1 = M[stack\_pointer];  stack\_pointer++;   * Previous activation records need to be accessed, so push/pop not sufficient.   + Treat stack as array with index off of stack pointer.   – Push and pop entire activation records. |
|  | **Illustrate type checking with necessary diagram (Nov/Dec 2016)**   * The compiler should report an error if an operator is applied to an incompatible operand. * Type checking can be performed without running the program. It is a static check as opposed to a dynamic check which is performed during run time.   Examples:   * Flow-of-Control: if control leaves a construct is there a place for it to go to? An example is a break statement in C not enclosed within a while, for, or, switch statement. * Uniqueness: Each variable should only be declared once. A function or procedure should only be defined once. Labels in a case statement should be unique. * Basic types are atomic types with no internal structure visible to the programmer. Pascal examples are: boolean, character, integer and real. Subrange types like 1….10 and enumerated types, like (violet, indigo, blue, green, yellow, orange, red) are also basic types. * Constructed types built from other types. Pascal examples are arrays, records, sets and pointers. * The type of a language construct is denoted by a type expression. Type expressions are basic types or constructions of type expressions. * Basic type expressions: Includes the basic types of the language like boolean, character, integer and real. * A special basic type expression is type\_error to signal the occurrence of a type check error. * Statements have no values and are assigned the basic type expression of void. * Type names are basic type expressions. * Array(I, T) is a type expression denoting the type of an array with elements of type T and index set I. * Cartesian product: If T1 and T2 are type expressions then T1 \* T2 is a type expression. * The type of a list is the Cartesian product of the types of its elements. * E.g. if a and b are of type real then a, b is of real \* real type. * Records: The type expression for a record with fields F1, F2, …, Fn of types T1, T2, …, Tn resply is :   record ((F1 \* T1) \* (F2 \* T2)\* … \* (Fn \* Tn)).   * Pointers: If T is a type expression then pointer(T) is the type expression for a pointer to an object of type T. * Functions: The type expression for a function is D 🡺 R where D is the type expression for the domain of the function and R is the type expression for the range of the function. If the function has no arguments then the domain type is void. * Procedures: A procedure does not return any value so it can be treated like a function where the type expression for the range is void. * The grammar is :   prog 🡪 decls; expr  decls 🡪 decls; decls | id : type  type 🡪 char | integer | array[num] of type | type  expr 🡪 literal | num | id | expr mod expr | expr [expr] | expr   * A prog has decls written before expr so the types of all declared identifiers can be saved in the symbol table before the expression is checked. A translation scheme to save the types of identifiers is:   prog 🡪 decls; expr  decls 🡪 decls; decls  decls 🡪 id : type {addtype(id.entry, T.type)}  type 🡪 char {T.type := char}  type 🡪 integer {T .type := integer}  type 🡪 type1 {T.type := pointer(T1.type)}  type 🡪 array [num] of type1 {T.type := array (1..num.val, T1.type)}   * We call a function lookup (e) to fetch the type of entry e in the symbol table. A translation scheme to type check expressions is :   expr 🡪 literal {E.type := char}  expr 🡪 num {E.type := integer}  expr 🡪 id {E.type := lookup (id.entry)}  expr 🡪 expr1 mod expr2 {E.type := if E1.type = integer and E2.type = integer  then integer else type\_error}  expr 🡪 expr1 [ expr2 ] {E.type := if E2.type = integer and E1.type = array (s,t)  then t else type\_error}  expr 🡪 expr1 {E.type := if E1.type = pointer (t) then t else type\_error}   * The type expression of a statement is void if it is correct or type\_error if a type error is found. * E.g. prog 🡪 decls; stmt   stmt 🡪 id := expr {S.type := if lookup (id.entry) = E.type then void else type\_error}  stmt 🡪 if expr then stmt1 {S.type := if E.type = boolean then S1.type else type\_error}  stmt 🡪 while expr do stmt1 {S.type := if E.type = boolean then S1.type else type\_error}  stmt 🡪 stmt1 ; stmt2 {S.type := if S1.type = void then S2.type else type\_error}   * Type checking of calls to no-argument functions. The statement a := b sets a to the value returned by function b if b is a function or to the value of b if b is a variable.   expr 🡪 id {E.type := if lookup(id.entry) = (void 🡺 t) or lookup (id.entry) = t then t else type\_error}   * Type checking of procedure statements   stmt 🡪 id (expr) {S.type := if E.type = e and lookup(id.entry) = (e 🡺 void) then void else type\_error}  stmt 🡪 id {S.type := if lookup (id.entry) = (void 🡺 void) then void else type\_error}   * Two standard types : integer and real * Four types of variables: integer, real, integer array and real array * The index sets of all arrays are integers. We won’t keep track of the limits in type checking. * Factors, terms and expressions maybe: integer, real or boolean * Each argument of a functinon or a procedure maybe: integer, real, integer array or real array * Functions return either integers or reals * The symbol table contains the names of variables, functions and procedures * Encoding of type expressions: the alphabet has 8 symbols representing integer, real , integer\_array, real\_array, boolen, void, type\_error and 🡺. * The type expression of an expression\_list or a parameter\_list is a string of symbols with one symbol for each element of the list. * The 🡺 symbol is used to encode functions and procedures: * The type expression for a real function of an integer and a real is: integer real🡺 real * The type expression for an integer function with no arguments is: void 🡺integer * The type expression for a procedure with an integer and a real argument is: integer real 🡺 void * The type expression for a procedure with no arguments is: void 🡺 void * The type expressions for functions and procedures in the symbol table could be stored in the name array |
|  | **A syntax directed translation scheme that takes string a’s b’s and c’s as input and produces an output the number of substrings in the input string that correspond to the pattern a(a|b)\*c+(a|b)\*b. For example the translation of the input string “abbcabcababc” is “3”. (Nov/Dec 2016)**  **(1) Write a context-free grammar that generates all strings of a’s, b’s and c’s**  **(2) Semantic attributes for the grammar symbols**  **(3) For each production of the grammar a set of rules for evaluation of the semantic attributes** |
|  | **Discuss specification of a simple type checker. (Ma/June,2016) (April/May,2017)**  **Static Checking**   * Check that the source program follows both the syntactic and semantic conventions of the source language * Examples of static check   + Type checks (incompatible operand)   + Flow-of-control checks (break statement)   + Uniqueness checks (uniquely declared identifier)   + Name-related checks   **2. Position of type checker**   * A type checker verifies that the type of a construct matches that expected by its context. * Type information gathered by a type checker may be needed when code is generated. * A symbol that can represent different operations in different context is said to be “overloaded”   A simple language  1) Grammar for source language   * + P→D;E   + D→D;D | id:T   + T →char | integer | array [num] of T | ^T   E →literal | num | id | E mod E | E[E] | E^ |
|  | **Construct a syntax directed definition for constructing a syntax tree for assignment statements.**  **S *🡪 id :=E***  ***E 🡪 E+E***  ***E 🡪 E\*E***  ***E 🡪 (E)***  ***E 🡪 id* (May/June,2016)** |
| **UNIT - V / PART –B C6660.5** | |
|  | **Explain the principle sources of optimization in detail. (May/June,2016) (April/May,2017) (Nov/Dec, 2017)**  Transformations can be  – Local : look within basic block  – Global : look across blocks  • Transformations should preserve function of program.  • Function-preserving transformations include  – Common sub expression elimination  – Copy propagation  – Dead-code elimination  – Constant-folding  ***Common Sub expression Elimination***  • Occurrence of expression E is called common sub expression if  – E was previously computed, and  – values of variables in E have not changed since previous Computation    ***Copy Propagation***  • Statement of form f := g is called a copy statement  • Idea isto use g instead of f in subsequent statements  • Doesn't help by itself, but can combine with other transformations to help eliminate code:    ***Dead-Code Elimination***  • Variable that is no longer live (subsequently used) is called dead.  • Copy propagation often turns copy statement into dead code:    ***Loop Optimizations***  • Biggest speedups often come from moving code out of inner loop  • Three techniques  – Code motion  – Induction-variable elimination  – Reduction in strength  ***Code Motion***  • Expression whose value doesn't change inside loop is called a loop-invariant  • Code motion moves loop-invariants outside loop    ***Induction Variables and Reduction in Strength***  • Variables that remain "in lock step" with each other inside a loop are called induction variables  • E.g., decreasing array byte-offset index by 4 as loop variable decreases by 1:  • Addition is like multiplication, "reduced in strength" (less costly)  • Exploit induction variables and reduction -in-strength to make loop code more efficient |
|  | **Discuss about the following:**  **i) Copy Propagation**  **ii) Dead-code Elimination**  **iii) Code motion.**  ***Copy Propagation***  • Statement of form f := g is called a copy statement  • Idea isto use g instead of f in subsequent statements  • Doesn't help by itself, but can combine with other transformations to help eliminate code:  \    ***Dead-Code Elimination***  • Variable that is no longer live (subsequently used) is called dead.  • Copy propagation often turns copy statement into dead code:    ***Code Motion***  • Expression whose value doesn't change inside loop is called a loop-invariant  • Code motion moves loop-invariants outside loop |
|  | **(i).Explain optimization of basic blocks. (April/May,2017)**  There are two types of basic block optimizations. They are :  Ø     Structure-Preserving Transformations  Ø     Algebraic Transformations    **Structure-Preserving Transformations:**  The primary Structure-Preserving Transformation on basic blocks are:    Ø     Common sub-expression elimination  Ø     Dead code elimination  Ø     Renaming of temporary variables  Ø     Interchange of two independent adjacent statements.    **Common sub-expression elimination:**  Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it’s referenced.    Example:    **a:** **=b+c**  **b:** **=a-d**  **c:** **=b+c**  **d:** **=a-d**    The 2nd and 4th statements compute the same expression: b+c and a-d   Basic block can be transformed to  **a:** **= b+c**  **b:** **= a-d**  **c:** **= a**  **d:** **= b**  **Dead code elimination:**   It is possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.  **Renaming of temporary variables:**   A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u. In this a basic block is transformed to its equivalent block called normal-form block.    **Interchange of two independent adjacent statements:**   • Two statements  **t1:=b+c**  **t2:=x+y**   can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.    **Algebraic Transformations:**    Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2\*3.14 would be replaced by 6.28.  The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments    **a :=b+c**  **e :=c+d+b**    the following intermediate code may be generated: **a :=b+c**    **t :=c+d e :=t+b**    Example:  x:=x+0 can be removed  x:=y\*\*2 can be replaced by a cheaper statement x:=y\*y    The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x\*y-x\*z as x\*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.  (ii)Explain redundant common sub-expression elimination.  Suppose that in some code, the following portion exists: a = f \* i ... b = f \* i If the value of the expression f \*i has not changed between the first and second assignment, then we are doing an extra computation that is unnecessary. In this case, the expression f \* i is called a common subexpression. We can eliminate common subexpressions to potentially reduce the running time of the program. That is, we transform the code into this: a = f \* i temp = a ... b = temp where temp is a fresh variable.  Common subexpression elimination depends on what is called available expressions. At any particular point in the program, there is some set of expressions that has been computed, and whose values have not subesquently changed. For example, after the execution of a = f \* i above, the expression f \* i is available. As the value of f \* i did not change before the execution of b = f \* i, it was still available. Because it was available, we could perform the transformation. Common subexpression elimination needs to know what expressions are available for use. The available expressions data-flow analysis determines, for every program point, which expressions are available. There must be an evaluation of the expression on every path from the entry to the program point, and none of the variables occurring in the expression are defined between the last such evaluation on a path and the point.  An occurrence of an expression in a program is a common subexpression if the expression is available. Common subexpression elimination is a transformation that removes the recomputations of common subexpressions and replaces them with uses of the saved values of those common subexpressions.  Originally, only local common subexpression elimination was originally being coded for addition to the Titanium compiler. The method being used was the standard algorithm in Muchnick's book. When analyzing each node, one must determine what variables are killed, and consequently what subexpressions were no longer available. Rather than attempting to write my own analysis of which variables were killed, this was accomplished by taking every assignment and determining what variables were defined by that assignment by looking at who used this definition, utilizing the def/use info previously computed. This, however, was skating the thin edge of the analysis, and Dan suggested that the def/use analysis could be used in a more direct way, which would then also provide global common subexpression elimination for free.  Given the code:  a = f \* i  ...  b = f \* i One wishes to determine if the expression f \* i, computed at the first statement, is available at the second statement. We can do this only if the value of f \* i has not changed, which we can conclude from whether f or i has changed between the two assignment statements. That is, if f or i has a definition that kills the respective variable between the two assignments, then the expression is not available. However, as it stands, the def/use information available is not amenable for use to determine if a variable x is killed between point a and point b. One would need to ask the question: is the definition of f \* i in the first statement the only definition at the use of f\* i? But expressions do not have definitions. So, modify the code above by inserting assignments to produce:  f = f  i = i  a = f \* i  ...  b = f \* i  At this point, we now can use the created definitions of f and i as the definition of the expression f \* i. Now the question becomes: are the definitions of f and i in the second statement the same as the definitions of f and i in the first?  The above test is sufficient for straight line code, but not for code where more than one computation of an expression can take place. For example:  In this case, one wants to know if every definition of f and i are from the generated assignments associated with some computation of the expression. If there is a definition of f or i that is not from an inserted assignment, then there exists a path on which f or i is killed and f \* i is not recomputed, i.e. the expression f \* i is not available. The particular computations that are associated with instances of inserted assignments where the definitions reach the second computation is the set of assignments that are involved in the common subexpression.  This winnowing is necessary, for not all computations of f \* i reach the statement under examination.  To find the subexpressions that are canidates for elimination, the control flow graph is searched. All assignments with expressions on the right hand side that might be eliminatable are stored in a hash table. Any expression stored in the table more than once is then instrumented for the analysis to determine if it can be eliminated. As expressions of no more than two operands are considered, the code increase is linear in the number of assignments instrumented.  Running the def/use analysis is quite expensive. In particular, one does not want to run it for every expression under examination. Thus, all useless assignments are inserted at one time, the def/use analysis is run, and then each expression is examined. However, this adds one additional wrinkle: all inserted assignments that are not for the particular subexpression under examination must be ignored.  Algorithm: For every location where an elimination is to be attempted: 1) For each variable in the expression: 1)a) search its definitions 1)b) if this is a real definition, we can not eliminate 1)c) if this definition is assiciated with an evaluation of the expression, save the location for later 1)d) if this definition is a different inserted assignment recurively look at its definitions 2)If we never found any real definitions, then this expression can be elinimated. Choose a fresh variable temp. At every saved location, add an instruction temp = lhs after the location, where lhs is the left hand side of the assignment at the saved location. 3) Replace the right hand side of the instruction at the location of elimination with a use of the new temporary variable temp. |
|  | **Write about data flow analysis of structural programs.**  Data-flow analysis derives information about the dynamic behavior of a program by only examining the static code.  **Liveness Analysis**  ***Definition***  – A variable is live at a particular point in the program if its value at that point will be used in the future (dead, otherwise).  ∴ To compute liveness at a given point, we need to look into the future Motivation: Register Allocation  – A program contains an unbounded number of variables  – Must execute on a machine with a bounded number of registers  – Two variables can use the same register if they are never in use at the same time (i.e, never  simultaneously live).  ∴ Register allocation uses liveness information |
|  | **Optimize the following code using various optimization techniques:**  **i=1,s=0;**  **for(i=1;i<=3;i++)**  **for(j=1;j<=3;j++)**  **c[i][j]=c[i][j]+a[i][j]+b[i][j];**  L1: t1 = i \* N  t2 = t1 + j  t3 = t2 \* 4  t4 = &c + t3  t12 = t1 + k  t13 = t12 \* 4  t14 = &a + t13  t21 = k \* N  t22 = t21 + j  t23 = t22 \* 4  t24 = &b + t23  t31 = \*t14 \* \*t24  \*t4 = \*t4 + t31  k = k + 1  if( k < N) goto L1  t1 = i \* N  t2 = t1 + j  t3 = t2 \* 4  t4 = &c + t3  L1: t12 = t1 + k  t13 = t12 \* 4  t14 = &a + t13  t21 = k \* N  t22 = t21 + j  t23 = t22 \* 4  t24 = &b + t23  t31 = \*t14 \* \*t24  \*t4 = \*t4 + t31  k = k + 1  **if( k < N) goto L1** |
|  | **(i)Explain the issues in design of code generator. (May/June,2016) (April/May,2017) (Nov/Dec, 2017)**  **(ii)Explain peephole optimization.**  **(i) Issues in the Design of Code generator**   * Memory management. * Instruction Selection. * Register Utilization (Allocation). * Evaluation order.   **1. Memory Management**  Mapping names in the source program to address of data object is cooperating done in pass 1 (Front end) and pass 2 (code generator).  Quadruples → address Instruction.  Local variables (local to functions or procedures ) are stack-allocated in the activation record while global variables are in a static area.  **2. Instruction Selection**  The nature of instruction set of the target machine determines selection.  -"Easy" if instruction set is regular that is uniform and complete.  Uniform: all triple addresses  all stack single addresses.  Complete: use all register for any operation.  If we don't care about efficiency of target program, instruction selection is straight forward.  For example, the address code is:  a := b + c  d := a + e  Inefficient assembly code is:  MOV b, R0 R0 ← b  ADD c, R0 R0 ← c + R0  MOV R0, a a ← R0  MOV a, R0 R0 ← a  ADD e, R0 R0 ← e + R0  MOV R0 , d d ← R0  Here the fourth statement is redundant, and so is the third statement if 'a' is not subsequently used.  **3. Register Allocation**  Register can be accessed faster than memory words. Frequently accessed variables should reside in registers (register allocation). Register assignment is picking a specific register for each such variable.  Formally, there are two steps in register allocation:  Register allocation (what register?)  This is a register selection process in which we select the set of variables that will reside in register.  Register assignment (what variable?)  Here we pick the register that contain variable. Note that this is a NP-Complete problem.  Some of the issues that complicate register allocation (problem).  1. Special use of hardware for example, some instructions require specific register.  2. Convention for Software:  For example  Register R6 (say) always return address.  Register R5 (say) for stack pointer.  Similarly, we assigned registers for branch and link, frames, heaps, etc.,  3. Choice of Evaluation order  Changing the order of evaluation may produce more efficient code.  This is NP-complete problem but we can bypass this hindrance by generating code for quadruples in the order in which they have been produced by intermediate code generator.  ADD x, Y, T1  ADD a, b, T2  is legal because X, Y and a, b are different (not dependent).  ***(*ii) peephole optimization**  Constant folding – Evaluate constant subexpressions in advance.  Strength reduction – Replace slow operations with faster equivalents.  Null sequences – Delete useless operations.  Combine operations – Replace several operations with one equivalent.  Algebraic laws – Use algebraic laws to simplify or reorder instructions.  Special case instructions – Use instructions designed for special operand cases.  Address mode operations – Use address modes to simplify code. |
|  | **Explain the simple code generator with a suitable example (May/June,2016)**  Code Generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.  Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the completed processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (the backend) needs to change from target to target.  **Example**  MOV R0 x,R0  MOV R1 y,R1  MUL R0,R1  MOV t1, R0  MOV R0 t1,R0  MOV R1 z,R1  ADD R0,R1  MOV t2, R0  MOV R0 x,R0  MOV R1 x,R1  ADD R0,R1  MOV x, R0  MOV R0 y,R0  MOV R1 y,R1  SUB R0,R1  MOV y, R0  OUT x  MOV z,y  OUT z |
|  | **Write detailed notes on Basic blocks and flow graphs.**  A graph representation of intermediate code.  Basic block properties   * The flow of control can only enter the basic block through the first instruction in the block. * No jumps into the middle of the block. * Control leaves the block without halting / branching (except may be the last instruction of the block). * The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.     **flow graphs**  A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets; jump targets start a block, and jumps end a block. Directed edges are used to represent jumps in the control flow. There are, in most presentations, two specially designated blocks: the entry block, through which control enters into the flow graph, and the exit block, through which all control flow leaves. |
|  | **Define a Directed Acyclic Graph. Construct a DAG and write the sequence of instructions for the expression a+a\*(b-c)+(b-c)\*d. (May/June 2014)**   * A representation to assist in code reordering. * Nodes are operations * Edges represent dependences * Nodes are labeled as follows: * Leaves with variables or constants – subscript 0 are used to distinguish initial value of the variable from other values. * Interior nodes with operators and list of variables whose values are computed by the node.   directed_acyclic_graph_2  **DAG for a+a\*(b-c)+(b-c)\*d** |
|  | **(i).Explain loops in flow graphs.**  A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.    **Dominators:**    In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.    Example:  \*In the flow graph below,    \*Initial node,node1 dominates every node. \*node 2 dominates itself    \*node 3 dominates all but 1 and 2. \*node 4 dominates all but 1,2 and 3.  \*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.    \*node 7 dominates 7,8 ,9 and 10. \*node 8 dominates 8,9 and 10.  \*node 9 and 10 dominates only themselves.  http://www.brainkart.com/media/extra/n3NyvMV.jpg  **Fig. 5.3(a) Flow graph (b) Dominator tree**    The way of presenting dominator information is in a tree, called the dominator tree, in which  •   The initial node is the root.  •   The parent of each other node is its immediate dominator.  •   Each node d dominates only its descendents in the tree.    The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n. In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.    \*\*\*  D(1)={1}  D(2)={1,2}  D(3)={1,3}  D(4)={1,3,4}  D(5)={1,3,4,5}  D(6)={1,3,4,6}  D(7)={1,3,4,7}  D(8)={1,3,4,7,8}  D(9)={1,3,4,7,8,9}  D(10)={1,3,4,7,8,10}  (ii).Explain Local optimization.  Local Optimizations: Algebraic Simplification  • Some statements can be deleted  x := x + 0  x := x \* 1  • Some statements can be simplified or converted to use faster operations:  Original Simplified  x := x \* 0 x := 0  y := y \*\* 2 y := y \* y  x := x \* 8 x := x << 3  x := x \* 15 t := x << 4; x := t - x  (on some machines << is faster than \*; but not on all!)  Local Optimization: Constant Folding  • Operations on constants can be computed at compile time.  • Example: x := 2 + 2 becomes x := 4.  • Example: if 2 < 0 jump L becomes a no-op. |
|  | **(i) Write the code generation algorithm using dynamic programming and generate code for the statement x=a/(b-c)-s\*(e+f) [Assume all instructions to be unit cost] (12)**  **(ii) What are the advantages of DAG representation? Give example.(4) (April/May 2015)**  **(i) Code Generation using dynamic programming**  Goal: Generate optimal code for broad class of register machines  **Machine Model:**   * k interchangeable registers r0, r1, . . . , rk−1. * Instructions are of the form ri := E, where E is an expression containing operators, registers, and memory locations (denoted M) * Every instruction has an associated cost, measured by C()   Cost Vector: C(E) = (c0 c1 · · · cr ) — it’s defined for an expression E, where:   * C0: cost of computing E into memory, with the use of unbounded number of regs * Ci: cost of computing E into a register, with the use of up to i regs   **(ii)Advantages of DAG representation and its example.**   * + - Determining the common sub-expressions.     - Determining which names are used inside the block and computed outside the block.     - Determining which statements of the block could have their computed value outside the block.   Simplifying the list of quadruples by eliminating the common sub-expressions and not performing. |
|  | **(i)Write the procedure to perform Register Allocation and Assignment with Graph Coloring.(8)**   * Two passes are used   + Target-machine instructions are selected as though there are an infinite number of symbolic registers   + Assign physical registers to symbolic ones     - Create a register-interference graph     - Nodes are symbolic registers and edges connects two nodes if one is live at a point where the other is defined.     - For example in the previous example an edge connects a and d in the graph   Use a graph coloring algorithm to assign registers.  **The Register Interference Graph**  • Two temporaries that are live simultaneously cannot be allocated in the same register  • We construct an undirected graph  – A node for each temporary  – An edge between t1 and t2 if they are live simultaneously at some point in the program  • This is the register interference graph (RIG)  – Two temporaries can be allocated to the same register if there is no edge connecting them  example:    E.g., b and c cannot be in the same register  • E.g., b and d can be in the same register  **(ii)Construct DAG and optimal target code for the expression *X=((a+b)/(b-c))-(a+b)\*(b-c)+f.***   * There is a node in the DAG for each of the initial values of the variables appearing in the basic block. * There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s. * Node N is labeled by the operator applied at s, and also attached to N is the list of variables for which it is the last definition within the block. * Certain nodes are designated output nodes. These are the nodes whose variables are live on exit from the block.       1.In this section we assume we are using an n-register machine with instructions of the form  o LD reg, mem  o ST mem, reg  o OP reg, reg, reg  to evaluate expressions.  2.An expression tree is a syntax tree for an expression.  3.Numbers, called Ershov numbers, can be assigned to label the nodes of an expression tree. A node gives the minimum number of registers needed to evaluate on a register machine the expression generated by that node with no spills. A spill is a store instruction that gets generated when there are no empty registers and a register is needed to perform a computation.  4.Algorithm to label the nodes of an expression tree  1. Label all leaves 1. 2. The label of an interior node with one child is the label of its child.  3. The label of an interior node with two children is the larger of the labels of its children if these labels are different; otherwise, it is one plus the label of the left child. |
|  | **Perform analysis of available expressions on the following code by converting into basic blocks and compute global common sub expression elimination.**  I:=0  A:=n-3  If i<a then loop else end  Label loop  B:=i\_4  E:=p+b  D:-m[c]  E:=d-2  F:=I-4  G:=p+f  M[g]:=e  I:=i+1  A:=n-3  Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.  main()  {  int x, y, z;  x = (1+20)\* -x;  y = x\*x+(x/y);  y = z = (x/y)/(x\*x);  }  straight translation:  tmp1 = 1 + 20 ;  tmp2 = -x ;  x = tmp1 \* tmp2 ;  tmp3 = x \* x ;  tmp4 = x / y ;  y = tmp3 + tmp4 ;  tmp5 = x / y ;  tmp6 = x \* x ;  z = tmp5 / tmp6 ;  y = z ;  Here is an optimized version, after constant folding andpropagation and elimination of common sub- expressions:  tmp2 = -x ;  x = 21 \* tmp2 ;  tmp3 = x \* x ;  tmp4 = x / y ;  y = tmp3 + tmp4 ;  tmp5 = x / y ;  z = tmp5 / tmp3 ;  ***y = z ;*** |
|  | **(i) Explain Loop optimization in details and apply it to the code (10)**  Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:   * **Invariant code :** A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration. * **Induction analysis :** A variable is called an induction variable if its value is altered within the loop by a loop-invariant value. * **Strength reduction :** There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x \* 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.   **(ii)What is the optimization technique applied on procedure calls? Explain with example .(6) (April/May 2015)**  Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform.  **Induction variable analysis**  If a variable in a loop is a simple linear function of the index variable, such as j := 4\*i + 1, it can be updated appropriately each time the loop variable is changed. This is a strength reduction, and also may allow the index variable's definitions to become dead code. This information is also useful for bounds-checking elimination and dependence analysis, among other things.  A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.  The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).  Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack. We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**. |
|  | **(i) Write an algorithm for constructing natural loop of back edge**  **(ii) Explain any four issues that crop up when designing a code generator (Nov/Dec 2016)**   * 1. **Natural loop**   We use dominators to find loops. Since loops contain a cycle, and a header dominates all the nodes in a loop, there must be a least one arc entering the header from a node in the loop. For this reason, we search for an arc whose head dominates its tail. This is called a back edge. Loops must have a back edge.  The natural loop of the back edge is defined to be the smallest set of nodes that includes the back edge and has no predecessors outside the set except for the predecessor of the header. Natural loops are the loops for which we find optimizations.  **Algorithm**  Finding the nodes in a natural loop  FOR every node n, find all m such that n DOM m  FOR every back edge tarrowh, i.e., for every arc such that h DOM t, construct the natural loop:  Delete h and find all nodes which can lead to t  These nodes plus h form the natural loop of tarrowh  pg158x13  **(ii)Issues in the Design of Code generator**   * Memory management. * Instruction Selection. * Register Utilization (Allocation). * Evaluation order.   **1. Memory Management**  Mapping names in the source program to address of data object is cooperating done in pass 1 (Front end) and pass 2 (code generator).  Quadruples → address Instruction.  Local variables (local to functions or procedures ) are stack-allocated in the activation record while global variables are in a static area.  **2. Instruction Selection**  The nature of instruction set of the target machine determines selection.  -"Easy" if instruction set is regular that is uniform and complete.  Uniform: all triple addresses  all stack single addresses.  Complete: use all register for any operation.  If we don't care about efficiency of target program, instruction selection is straight forward.  For example, the address code is:  a := b + c  d := a + e  Inefficient assembly code is:  MOV b, R0 R0 ← b  ADD c, R0 R0 ← c + R0  MOV R0, a a ← R0  MOV a, R0 R0 ← a  ADD e, R0 R0 ← e + R0  MOV R0 , d d ← R0  Here the fourth statement is redundant, and so is the third statement if 'a' is not subsequently used.  **3. Register Allocation**  Register can be accessed faster than memory words. Frequently accessed variables should reside in registers (register allocation). Register assignment is picking a specific register for each such variable. Formally, there are two steps in register allocation:  Register allocation (what register?)  This is a register selection process in which we select the set of variables that will reside in register.  Register assignment (what variable?)  Here we pick the register that contain variable. Note that this is a NP-Complete problem.  Some of the issues that complicate register allocation (problem).  1. Special use of hardware for example, some instructions require specific register.  2. Convention for Software:  For example  Register R6 (say) always return address.  Register R5 (say) for stack pointer.  Similarly, we assigned registers for branch and link, frames, heaps, etc.,  **4. Choice of Evaluation order**  Changing the order of evaluation may produce more efficient code.  This is NP-complete problem but we can bypass this hindrance by generating code for quadruples in the order in which they have been produced by intermediate code generator.  ADD x, Y, T1  ADD a, b, T2  is legal because X, Y and a, b are different (not dependent). |
|  | **Explain global data flow analysis with necessary equations (Nov/Dec 2016)**  **Consider the following language:**  S ::= id:=expression | S;S | if expression then S else S | do S while expression  We define the following equations to compute reaching definitions. In these  equations we compute reaching definitions for statements instead of blocks.  We use the following terms:  • gen[S] is the set of definitions “generated by S”.  • kill[S] is the set of definitions “killed” by S.  • in[S] is the set of definitions reaching S (or the top of S)  • out[S] is the set of definitions that reach the bottom of S.  1.When S has the form a :=expression and label d:  gen[S] = {d}  kill[S] = Da-{d}. Da is the set of all definitions of a.  out[S] = gen[S] + (in[S]-kill[S])  2.When S is of the form S1; S2:  gen[S] = gen[S2]+(gen[S1]-kill[S2])  kill[S] = kill[S2]+(kill[S1]-gen[S2])  in[S1] = in[S]  in[S2] = out[S1]  out[S] = out[S2]  3.When S is of the form if ... then S1 else S2  gen[S]=gen[S1]+gen[S2]  kill[S]=kill[S1]∩kill[S2]  in[S1] = in[S]  in[S2] = in[S]  out[S] = out[S1]+out[S2]  4.When S is of the form do S1 while ...:  gen[S] = gen[S1]  kill[S] = kill[S1]  in[S1] = in[S]+gen[S1]  out[S] = out[S1] |
|  | **Explain about parameter passing mechanism (April/May,2017)** Parameter Passing The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program. r-value The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable. l-value The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left hand side of an assignment operator.  For example:  day = 1;  week = day \* 7;  month = 1;  year = month \* 12;  From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.  For example:  7 = x + y;  is an l-value error, as the constant 7 does not represent any memory location. Formal Parameters Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function. Actual Parameters Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.  **Example:**  fun\_one()  {  int actual\_parameter = 10;  call fun\_two(int actual\_parameter);  }  fun\_two(int formal\_parameter)  {  print formal\_parameter;  }  Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address. Pass by Value In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure’s activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters. Pass by Reference In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location. Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value. Pass by Copy-restore This parameter passing mechanism works similar to ‘pass-by-reference’ except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters.  **Example:**  int y;  calling\_procedure()  {  y = 10;  copy\_restore(y); //l-value of y is passed  printf y; //prints 99  }  copy\_restore(int x)  {  x = 99; // y still has value 10 (unaffected)  y = 0; // y is now 0  }  When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference. Pass by Name Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference |
|  | **Construct the DAG for the following Basic Block. (April/May,2017)**   1. **t1: =4\*i** 2. **t2: =a [t1]** 3. **t3: =4\*i** 4. **t4:=b[t3]** 5. **t5: =t2\*t4** 6. **t****6: =prod +t**[**5**](http://notes.pmr-insignia.org/) 7. **prod := t6** 8. **t7:=i+1** 9. **i:=t7** 10. **if i<=20 goto (1)** |

**------**