PCS6I102 COMPILER DESIGN

Module – I (16 Hours)

Introduction: Overview and Phases of compilation.
Lexical Analysis: Non-Deterministic and Deterministic Finite Automata (NFA & DFA), Regular grammar, Regular expressions and Regular languages, Design of a Lexical Analyzer as a DFA, Lexical Analyzer generator.
Syntax Analysis: Role of a Parser, Context free grammars and Context free languages, Parse trees and derivations, Ambiguous grammar.
Top Down Parsing: Recursive descent parsing, LL (1) grammars, Non-recursive Predictive Parsing, Error reporting and Recovery.
Bottom Up Parsing: Handle pruning and shift reduces Parsing, SLR parsers and construction or SLR parsing tables, LR(1) parsers and construction of LR(1) parsing tables, LALR parsers and construction of efficient LALR parsing tables, Parsing using Ambiguous grammars, Error reporting and Recovery,Parser generator.

Module – II (08 Hours)

Intermediate Code Generation: DAG for expressions, Three address codes - Quadruples and Triples, Types and declarations, Translation of Expressions, Array references, Type checking and Conversions, Translation of Boolean expressions and control flow statements, Back Patching, Intermediate Code Generation for Procedures.

Module – III (08 Hours)

Code Generation: Factors involved, Registers allocation, Simple code generation using STACK Allocation, Basic blocks and flow graphs, Simple code generation using flow graphs.

Code Optimization: Objective, Peephole Optimization, Concepts of Elimination of local common sub-expressions, Redundant and un-reachable codes, Basics of flow of control optimization.

Module – IV (08 Hours)

Run Time Environment: Storage Organizations, Static and Dynamic Storage Allocations, STACK Allocation, Handlings of activation records for calling sequences.
Syntax Directed Translation: Syntax Directed Definitions (SDD), Inherited and Synthesized Attributes, Dependency graphs, Evaluation orders for SDD, Semantic rules, Application of Syntax Directed Translation.
Symbol Table: Structure and features of symbol tables, symbol attributes and scopes.

Text Books:

1. Compilers – Principles, Techniques and Tools, A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Pearson.
2. Compiler Design, K. Muneeswaran, Oxford University Press

Reference Books:

1. Compiler Design, S. Chattopadhyay, PHI.
2. Modern Compiler Design, D. Galles, Pearson Education.
3. Advanced Compiler Design & Implementation, S. S. Muchnick, Morgan Kaufmann.
4. Compiler Design in C, A. I. Holub, PHI

COMPILER DESIGN LABORATORY

This lab is divided in to two parts namely part 1 and part 2. All programs in part 1 must be written using C/C++. Programs related to lexical analyzer and parser must use Flex(Fast Lex) and Yacc available in all modern versions of UNIX and Linux distributions. For part 2, a simulator JFLAP is required to be installed. JFLAP works much like a black box and used to hide all implementation details and thus should only be used after students. JFLAP is available online at http://www.jflap.org/.

PART 1

1. Using JFLAP, create a DFA from a given regular expression. All types of error must be checked during the conversion.
2. Read a regular expression in standard form and check its validity by converting it to postfix form. Scan a string and check whether the string matches against the given regular expression or not.
3. (Tokenizing). A programs that reads a source code in C/C++ from an unformatted file and extract various types of tokens from it (e.g. keywords/variable names, operators, constant values).
4. Read a regular expression in its standard form and find out an Ɛ-NFA from it. Need to use adjacency list data structure of graph to store NFA. Thompson's construction needs to be used too. [2 labs]
5. Evaluate an arithmetic expression with parentheses, unary and binary operators using Flex and Yacc.[Need to write yylex() function and to be used with Lex and yacc.]
6. (Tokenizing) Use Lex and yacc to extract tokens from a given source code.

PART 2

7. Write a suitable data structure to store a Context Free Grammar. Prerequisite is to eliminate left recursion from the grammar before storing. Write functions to find FIRST and FOLLOW of all the variables.[May use unformatted file / array to store the result].
8. Using JFLAP create LL(1) parse table for a given CFG and hence Simulate LL(1) parsing.
9. Using JFLAP create SLR(1) parse table for a given grammar. Simulate parsing and output the parse tree proper format.