# COMPILER DESIGN

**V Semester: CSE / IT**

| Course Code | Category | Hours / Week | | | Credits | Maximum Marks | | |
|---|---|---|---|---|---|---|---|---|
| | | L | T | P | C | CIA | SEE | Total |
| AIT004 | Core | 3 | 1 | - | 4 | 30 | 70 | 100 |
| Contact Classes: 45 | Tutorial Classes: 15 | Practical Classes: Nil | | | | Total Classes: 60 | | |

## OBJECTIVES:

**The course should enable the students to:**

I. Apply the principles in the theory of computation to the various stages in the design of compilers.
II. Demonstrate the phases of the compilation process and able to describe the purpose and operation of each phase.
III. Analyze problems related to the stages in the translation process
IV. Exercise and reinforce prior programming knowledge with a non-trivial programming project to construct a compiler.

## COURSE OUTCOMES:

I. Understand the various phases of compiler and design the lexical analyzer.Demonstrate the phases of the compilation process and able to describe the purpose and operation of each phase.
II. Explore the similarities and differences among various parsing techniques and grammar transformation techniques
III. Analyze and implement syntax directed translations schemes and intermediate code generation.
IV. Describe the concepts of type checking and analyze runtime allocation strategies.
V. Demonstrate the algorithms to perform code optimization and code generation.

## COURSE LEARNING OUTCOMES:

Students, who complete the course, will have demonstrated the ability to do the following:

1. Define the phases of a typical compiler, including the front and backend.
2. Recognize the underlying formal models such as finite state automata, push-down automata and their connection to language definition through regular expressions and grammars.
3. Identify tokens of a typical high-level programming language; define regular expressions for tokens and design and implement a lexical analyzer using a typical scanner generator.
4. Explain the role of a parser in a compiler and relate the yield of a parse tree to a grammar derivation.
5. Apply an algorithm for a top down or a bottom-up parser construction; construct a parser for a given context – free grammar.
6. Demonstrate Lex tool to create a lexical analyzer and Yacc tool to create a parser.
7. Understand syntax directed translation schemes for a given context free grammar.
8. Implement the static semantic checking and type checking using syntax directed definition(SDD) and syntax directed translation(SDT).
9. Understand the need of intermediate code generation phase in compilers..
10. Write intermediate code for statements like assignment, conditional, loops and functions in high level language.
11. Explain the role of semantic analyzer and type checking; create a syntax-directed and an annotated parse tree; describe the purpose of a syntax tree.
12. Students will be able to design syntax directed translation schemes for a given context free grammar
13. Explain the role of different types of runtime environments and memory organization for implementation of programming languages.

14. Differentiate static vs. dynamic storage allocation and the usage of activation records to manage program modules and their data.
15. Understand the role of symbol table data structure in the construction of compiler.
16. Learn the code optimization techniques to improve the performance of a program in terms of speed & space.
17. Implement the global optimization using data flow analysis such as basic blocks and DAG.
18. Understand the code generation techniques to generate target code.
19. Design and implement a small compiler using a software engineering approach.
20. Apply the optimization techniques to intermediate code and generate machine code.

| UNIT-I | INTRODUCTION TO COMPILERS AND PARSING | Hours: 10 |
|---|---|---|

Introduction to compilers: Definition of compiler, interpreter and its differences, the phases of a compiler,role of lexical analyzer, regular expressions, finite automata, from regular expressions to finite automata,pass and phases of translation, bootstrapping, LEX-lexical analyzer generator; Parsing: Parsing, role of parser, context free grammar, derivations, parse trees, ambiguity, elimination of left recursion, left factoring, eliminating ambiguity from dangling-else grammar, classes of parsing, top-down parsing:backtracking, recursive-descent parsing, predictive parsers, LL(1) grammars.

| UNIT-II | BOTTOM-UP PARSING | Hours: 10 |
|---|---|---|

Bottom-up parsing: Definition of bottom-up parsing, handles, handle pruning, stack implementation of shift- reduce parsing, conflicts during shift-reduce parsing, LR grammars, LR parsers-simple LR, canonical LR and Look Ahead LR parsers, error recovery in parsing, parsing ambiguous grammars, YACC-automatic parser generator.

| UNIT-III | SYNTAX-DIRECTED TRANSLATION AND INTERMEDIATE CODE GENERATION | Hours: 08 |
|---|---|---|

Syntax-directed translation: Syntax directed definition, construction of syntax trees, S-attributed and Lattributed definitions, translation schemes, emitting a translation.
Intermediate code generation: Intermediate forms of source programs– abstract syntax tree, polish notation and three address code, types of three address statements and its implementation, syntax directed translation into three-address code, translation of simple statements, Boolean expressions and flow-of control statements

| UNIT-IV | TYPE CHECKING AND RUN TIME ENVIRONMENT | Hours: 09 |
|---|---|---|

Type checking: Definition of type checking, type expressions, type systems, static and dynamic checking of types, specification of a simple type checker, equivalence of type expressions, type conversions, overloading of functions and operators; Run time environments: Source language issues, Storage organization, storage- allocation strategies, access to nonlocal names, parameter passing, symbol tables, and language facilities for dynamic storage allocation.

| UNIT-V | CODE OPTIMIZATION AND CODE GENERATOR | Hours: 08 |
|---|---|---|

Code optimization: The principle sources of optimization, optimization of basic blocks, loops in flow graphs, peophole optimization; Code generator: Issues in the design of a code generator, the target machine, runtime storage management, basic blocks and flow graphs, a simple code generator, register allocation and assignment, DAG representation of basic blocks.

## Text Books:

1. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, -Compilers-Principles, Techniques and Tools‖, Pearson Education, Low Price Edition, 2004

**Reference Books:**


1. Kenneth C. Louden, Thomson, ―Compiler Construction– Principles and Practice‖, PWS Publishing 1st Edition ,1997
2. Andrew W. Appel, ―Modern Compiler Implementation C‖, Cambridge University Press, Revised Edition, 2004.
3. Andrew W. Appel, Modern Compiler Implementation C, Cambridge University Press, 2004.