Course Title	Languages and Compilation				
Course Code	ACSC371				
Course Type	Elective				
Level	BSc (Level 1)				
Year / Semester	3rd / 4th (Fall/Spring)				
Teacher's Name	Stephania Loizidou, Chrysostomos Chrysostomou				
ECTS	6	Lectures / week	3	Laboratories/week	0
Course Purpose	This course aims to familiarize students with the concepts and principles underlying the field of languages and compilation, and to enable students develop the skills related to the theory and practice of compiler design. The role of basic principles and techniques that form the construction of compiler is emphasized through practical work carried out by designing, implementing, and testing a compiler.				
Learning Outcomes	By the end of the course, the students are expected to: 1. outline the trends in programming languages and machine architecture that are shaping compilers; 2. recall the relationship between compiler design and computer-science theory; 3. examine, analyze and assess the basic principles and techniques that form the construction of compiler; 4. identify the difficulties in developing modern compilers, describe and evaluate the techniques used to support such features; 5. assess the theory and practice of compiler design; 6. design, construct, implement, and test/verify a compiler for a simple programming language.				
Prerequisites	ACSC288, ACSC300, ACSC382	Co-requisites	None		
Course Content	• **Introduction to Programming Languages:** Why programming languages? Why different languages? Main programming languages paradigms. Goals and trade-offs. Domains of application. Comparison of programming languages. Historical development of Programming Languages. • **Introduction to Compiling:** What is a compiler? Language processors. High level overview of the structure of a typical compiler (the analysis-synthesis model of compilation, Phases of a compiler: Lexical analysis, Syntax analysis, Semantic analysis, Intermediate code generation, Code optimization, Code generation, Symbol-table management, The grouping of phases into passes, Compiler-construction tools). The evolution of programming languages. The science of building a				

compiler. Applications of compiler technology.

- **Lexical Analysis:** The role of the lexical analyzer. Lexical analysis versus parsing. Tokens, patterns, and lexemes. Attributes for tokens. Lexical errors. Input buffering. Alphabet, Languages, and Strings definitions. Operations on languages (union, concatenation, kleene closure, positive closure). Regular expressions. Regular definitions. Transition diagrams. Finite automata (NFA, DFA). From regular expressions to automata. The lexical-analyzer generator tool Lex.
- **Syntax Analysis:** The role of the parser. Syntax error handling. Error-recovery strategies. Context-free grammars (Definition of terminals, nonterminals, start symbol, and productions. Notational Conventions, Derivations, Parse trees. Ambiguity. Verifying the language generated by a grammar. Context-free grammars versus regular expressions). Writing a grammar (Eliminating ambiguity. Elimination of left recursion. Left factoring). Top-down parsing (Recursive-descent parsing. FIRST and FOLLOW. LL(1) grammars. Nonrecursive predictive parsing. Error recovery in predictive parsing). Bottom-up parsing (Reductions. Handle pruning. Shift-reduce parsing. Conflicts during shift-reduce parsing). LR(k) parsers (Items and the LR(0) automaton. The LR-parsing algorithm. Constructing SLR-parsing tables. Viable prefixes). The canonical-LR and LALR parsers. Constructing LALR parsing tables. Using ambiguous grammars (precedence and associatively to resolve conflicts. The “Dangling-Else” ambiguity. Error recovery in LR parsing). Parser Generator tool yacc.
- **Syntax-Directed Translation:** Syntax-directed definition (SDD). Inherited and synthesized attributes. Dependency graphs. “S-attributed” and “L-attributed” classes of SDD’s. Applications of syntax-directed translation (Construction of syntax trees. The structure of a Type). Syntax-directed translation (SDT) schemes (Postfix translation schemes, parser-stack implementation of postfix SDT’s, SDT’s with actions inside productions, eliminating left recursion from SDT’s, SDT’s for L-attributed definitions), Implementing L-attributed SDD’s.
- **Type Checking:** Rules for type checking. Static and dynamic checking of types. Flow-of-control checks. Uniqueness checks. Name-related checks. Type expressions. Error recovery. Type conversions. Overloading of functions and operators.
- **Intermediate-Code Generation:** Intermediate languages. Variants of syntax trees (Directed acyclic graphs). Three-Address code. Types of three-address statements. Syntax-directed translation into three-address code. Implementation of three-address statements. Declarations. Control -flow translation of Boolean expressions. Short-circuit code. Backpatching. Intermediate code for procedures.
- **Code Generation:** Issues in the design of a code generator. The target language. Addresses in the target code. Basic blocks and flow graphs. A simple code generator. Register allocation and assignment. Peephole optimization.
- **Code Optimization:** Criteria for code-improving transformations. Causes of redundancy. Global common subexpressions. Copy propagation. Dead-code elimination. Code motion. Induction variables and reduction in strength.

	• **Exception handling:** The need for exception handling and abstraction from error handling. Exception and error handling mechanisms.
Teaching Methodology	Students are taught the course through lectures by means of computer presentations. Lectures are supplemented with demonstration of useful tools for generating lexical- and syntax-analyzers. Homework is provided consisting of practical problems to help students familiarizing with the theory and practice of compiler design. Central to the course is the project, where students are required to design, implement, and test a compiler for a simple programming language, aiming to help students develop practical skills by illustrating the concepts taught at lectures. Lecture/Coursework notes and presentations are available for students to use in combination with the textbooks and references, through the university's e-learning platform.
Bibliography	Textbook: • A.V. Aho, M.S. Lam, R. Sethi and J.D. Ullman, ***Compilers: Principles, Techniques***, and Tools - Pearson, 2nd Ed., 2007 [paper format] - Pearson, 2nd Ed., 2013 [eText format] References: • R. Toal, R. Rivera, A. Schneider, and E. Choe, ***Programming Language Explorations***, Chapman and Hall/CRC, 1st Ed., 2016 • K. Cooper, and L. Torczon, ***Engineering: A Compiler***, Morgan Kaufmann, 2nd Ed., 2011 • D. Brown, J. Levine, T. Mason, ***lex & yacc***, O'Reilly Media, 2nd Ed., 2012 • Notes/Manuals on Lexical-analyzer (lex/flex) and Parser (yacc/bison) generator tools – all available on the university's e-learning platform
Assessment	The assessment of the course includes one written test and a final written exam with problem-solving practical questions. Homework and project work are provided to help students familiarizing with the theory and practice of compiler design. The weights for each assessment component are: • Homework: 5% • Project: 20% • Test: 15% • Final Exam: 60%
Language	English