# Compiler Principles

1. **Course number and name**: 020PCOES4/020CPRES4 Compiler Principles

2. **Credits and contact hours**: 4 ECTS credits, 2x1:15 contact hours (course + lab)

3. **Name of course coordinator**: Maroun Chamoun

4. **Instructional materials**: Handouts posted on the Web

5. **Specific course information**
   a. **Catalog description**:
   Introduction to compilers – Lexical analysis: A language for specifying lexical analyzers, Finite automata, Design of a lexical analyzer generator, LEX tool. Algebraic grammar and pushdown automata - Syntax analysis: Top-down parsing and LL parsers, Bottom-up parsing and LR parsers, Parser generators and YACC tool – Semantic analysis: Syntax-directed definitions, Bottom-up evaluation, Top-down translation – Intermediate code generation: Three-address code, code optimization.

   b. **Prerequisites*:*** None

   c. **Required** for CCE Software Engineering Option students; **Selected Elective** for students in the CCE Artificial Intelligence and Telecommunication Networks Options.

6. **Educational objectives for the course**
   The primary goal of this course is to develop an understanding of the operation of compilers and the development and specification of computer-based languages. The course pulls together threads from underlying theory, most notably from logic and from data structures and algorithms, and builds on these a practical exercise in which students create a compiler of their own using commonly available compiler development tools.

   a. **Specific outcomes of instruction:**
   - Develop the notion of programming: data structures and advanced algorithms.
   - Become familiar with the development and maintenance of complex software.
   - Understand the compilation process and know how to implement the elements of compilation (lexical analysis, syntactic analysis) as well as operational semantics, interpreter and abstract machine.
   - Apply concepts of formal languages and finite-state machines to the translation of computer languages.
   - Identify the compiler techniques, methods, and tools that are applicable to other software applications.
   - Describe the challenges and state-of-the-practice of compiler theory and practice.

- Use compilation techniques to adapt a given language to a particular application as a data processing tool.
- Approach and use a new programming language.
- Implement a compiler for a simple language.

**b. PI addressed by the course:**

| **PI** | 1.1 | 1.2 | 1.3 |
|---|---|---|---|
| **Covered** | x | x | x |
| **Assessed** | x | x | x |

**7. Topics and approximate lecture hours**:

- Language translators: compilers and interpreters. Bootstrapping a compiler. The structure of a compiler: lexical analysis, parsing, semantic analysis, intermediate code generation, register allocation, global optimization. (2 Lectures)
- Lexical scanning: Token classes, keyword recognition, minimizing the code-per-character cost of scanning, scanning numeric literals and string literals. The interface between the scanner and the parser. Formalism: regular grammars, regular languages, Finite State Automata (FSA), automatic generation of lexical scanners. Hand-written vs. automatically generated scanners. Lex. (4 Lectures)
- Lab: Lexical scanning using Deterministic FSA. Introduction to Lex. Lexical scanning with Lex (3:45 hours)
- Parsing. Abstract syntax vs. concrete syntax. Grammars and the formal specification of certain aspects of programming languages. Top-down parsing and recursive descent. Automatic parser construction. FIRST and FOLLOW functions. LL(1) parsers. Bottom-up parsing through LR parsers. Conflicts in LR grammars and how to resolve them. SLR, LR(k), and LALR parsers. Yacc (7 Lectures)
- Lab: Parsing manually using LL parser. Automatic Parsing with Yacc (3:45 hours).
- Semantic analysis: attributes and their computation, tree-traversals, visibility and name resolution. Inherited attributes and symbol tables. Name resolution in block-structured languages. Type checking. Type systems, varieties of strong typing, overload resolution, polymorphism and dynamic dispatching. Type-checking and type inference, unification. (3 Lectures)
- Intermediate code generation: control structures, expressions, simple register allocation. Aggregates and other high-level constructs. (2 Lectures)
- Optimization: data flow analysis, Single-Assignment form. (1 Lecture)
- Lab: Writing a simple preprocessor using Lex and Yacc. (3:45 hours)