MODULE DESCRIPTION FORM

نموذج وصف المادة الدراسية

Module Information معلومات المادة الدراسية					
Module Title	Compilers			**Module Delivery**	
Module Type	**Core**			☒**Theory** ☒**Lecture** ☒**Lab** ☐**Tutorial** ☐**Practical** ☐**Seminar**	
Module Code	**MU010602203**				
ECTS Credits	**5**				
SWL (hr/sem)	**125**				
Module Level	UGII		**Semester of Delivery**	4	
Administering Department	Computer Science		**College**	College of Science	
Module Leader			**e-mail**		
Module Leader's Acad. Title	lecturer		**Module Leader's Qualification**		
Module Tutor			**e-mail**		
Peer Reviewer Name			**e-mail**		
Scientific Committee Approval Date	22 / 4 /2024		**Version Number**	1.0	

Relation with other Modules العلاقة مع المواد الدراسية الأخرى			
Prerequisite module	Theory of Computation (2)	**Semester**	1
Co-requisites module	None	**Semester**	

Module Aims, Learning Outcomes and Indicative Contents
أهداف المادة الدراسية ونتائج التعلم والمحتويات الإرشادية

Module Objectives أهداف المادة الدراسية	The module objectives of a Compiler course typically cover the fundamental principles, techniques, and tools involved in the design and implementation of a compiler. Below are some common objectives you might find in such a course: **Understanding the Role of Compilers**, explain the role of a compiler in the software development process. Differentiate between various types of compilers and interpreters. Understand the phases of compilation: lexical analysis, syntax analysis, semantic analysis, optimization, and code generation. **Practical Compiler Implementation** Apply the theoretical knowledge to implement a simple working compiler for a small programming language. Test and debug the compiler to ensure it correctly translates source code into executable code. Explore the challenges and considerations in real-world compiler design. These objectives aim to equip students with both the theoretical understanding and practical skills needed to design and build a basic compiler.
Module Learning Outcomes مخرجات التعلم للمادة الدراسية	*At the end of the course, students are expected to learn:* • Students will have the experience of designing and constructing compiler. • Developing the Students ability to design and implement compilers for specific purposes. • Acquire familiarity with basic data structures used in compiler construction such as abstract syntax trees, symbol tables, three-address code, and stack machines.
Indicative Contents المحتويات الإرشادية	The indicative contents of a Compiler course typically encompass a range of topics that cover the theoretical foundations, practical techniques, and tools necessary for understanding and building compilers. Below is an outline of the typical content: **UNIT I: [** 13 hrs.] **Introduction to Compilers**: Overview of Compilers Definition, functions, and importance. History of compiler development. Difference between compilers and interpreters. Structure of a typical compiler: Frontend, middle-end, and backend. Compilation process and its phases. **UNIT-II**: [18 hrs.] **Lexical Analysis**: Role of the lexical analyzer (lexer). Tokens, lexemes, and patterns. Regular expressions and finite automata. Construction of lexical analyzers. Transition diagrams, state machines. Tools for lexical analysis: Lex/Flex. Handling errors in lexical analysis. **UNIT-III**: [19 hrs.] **Syntax Analysis:** Role of the syntax analyzer (parser). Context-free grammars (CFGs). Parse trees and abstract syntax trees (ASTs). Parsing techniques: Top-down parsing (e.g., Recursive Descent, LL parsing), and Bottom-up parsing (e.g., LR parsing, SLR, LALR).

	Construction of parsers: Parser generators like Yacc/Bison. Error handling and recovery in parsing. **UNIT-IV:** [10 hrs.] **Semantic Analysis:** Purpose of semantic analysis. Symbol tables: Construction and management. Type checking and type systems. Attribute grammars and syntax-directed translation. Semantic error detection and reporting. **UNIT V (12)** **Intermediate Code Generation**: Three-address code (TAC), Quadruples, Triples **Code Optimization**: Local vs. global optimizations. Data flow analysis. optimization techniques. **Code Generation**. **Error Handling and Recovery**

Learning and Teaching Strategies استراتيجيات التعلم والتعليم	
Strategies	**Learning Strategies:** Learning strategies for a Compiler course are essential for mastering the complex concepts and practical skills required to design and implement compilers. Here are some effective strategies: **1. Understand the Big Picture** • Conceptual Framework: Start by understanding the overall structure of a compiler and how each phase (lexical analysis, syntax analysis, semantic analysis, etc.) fits into the larger process. • Flow of Compilation: Map out the flow from source code to executable code, identifying the role of each component. **2. Start with Theoretical Foundations** • Study Formal Languages and Automata: Gain a solid grasp of regular expressions, context-free grammars, and finite automata, as these form the basis of lexical and syntax analysis. • Mathematical Models: Engage with the mathematical models behind parsing techniques and type checking. **3. Hands-on Practice** • Build Simple Components: Begin by implementing basic components like a lexical analyzer and gradually move on to more complex ones like parsers and code generators. • Use Compiler Construction Tools: Get familiar with tools like Lex and Yacc or ANTLR. Start with small projects to see how these tools

automate parts of the compilation process.

4. Incremental Learning

- Phase-by-Phase Implementation: Focus on implementing one phase of the compiler at a time. For example, start with the lexer, then move on to the parser, semantic analyzer, and so on.
- Iterative Testing: Continuously test each phase with small examples to ensure correctness before moving on to the next phase.

5. Collaborative Learning

- Study Groups: Join or form study groups to discuss concepts, solve problems, and share insights. Explaining concepts to peers is a powerful way to reinforce your understanding.
- Pair Programming: Work with a partner on coding assignments to learn from each other's strengths and different problem-solving approaches.

6. Visualization Techniques

- Draw Parse Trees and Automata: Visualize how parsers and finite automata work by drawing parse trees and state machines for different inputs.
- Diagram Compilation Phases: Use flowcharts and diagrams to represent the flow of data and control between different phases of the compiler.

7. Apply Concepts to Real-world Languages

- Case Studies: Analyze the compilers of existing programming languages like C, Java, or Python to see how theoretical concepts are applied in real-world scenarios.
- Dissect Open-source Compilers: Explore open-source compiler projects like GCC, LLVM, or the Java Compiler to understand how professional-grade compilers are built.

8. Iterative Project Work

- Start Small: Begin with simple projects like building a calculator or a basic interpreter for a mini-language.
- Extend Projects: Gradually add features like type checking, optimization, or code generation to your projects, reflecting the stages of a full-fledged compiler.
- Capstone Project: Work on a comprehensive project where you build a compiler for a simple programming language from scratch.

9. Seek Feedback and Continuous Improvement

- Code Reviews: Regularly seek feedback on your code from instructors or peers. Participate in code reviews to learn best practices and optimization techniques.
- Refactor Code: Revisit and refactor your code as you learn more advanced techniques to make it more efficient and maintainable.

10. Use Online Resources and Documentation

- Compiler Textbooks: Refer to standard textbooks like "Compilers: Principles, Techniques, and Tools" (the Dragon Book) for in-depth understanding.
- Online Tutorials and Courses: Leverage MOOCs, video lectures, and online tutorials that break down complex topics into manageable segments.
- Documentation: Frequently refer to documentation for tools like Lex, Yacc, or LLVM to understand their capabilities and limitations.

11. Practice Problem-Solving

- Work on Compiler Design Problems: Solve problems related to grammar design, syntax tree construction, and optimization techniques.
- Participate in Competitions: Engage in coding competitions or challenges focused on compilers and language design to apply your knowledge in a competitive environment.

12. Reflect and Revise

- Regular Reviews: Periodically review your notes and revisit key concepts to reinforce your understanding.
- Self-assessment: Test your knowledge with quizzes, exercises, and by trying to explain concepts without referencing your notes.

These learning strategies will help you build a strong foundation in compiler design, enabling you to handle both theoretical and practical challenges.

Teaching Strategies:

Teaching a Compiler course effectively requires a combination of strategies that address both the theoretical concepts and practical skills involved. Here are some teaching strategies tailored for a Compiler course:

1. Start with a Conceptual Overview

- Big Picture Introduction: Begin the course by providing an overview of the entire compilation process. Explain the role of each phase (lexical analysis, syntax analysis, semantic analysis, etc.) and how they work together to translate source code into machine

code.
- Relate to Real-world Examples: Use familiar programming languages and their compilers (e.g., Java, C, Python) as examples to illustrate how compilers work in real-world applications.

2. Sequential and Incremental Learning

- Phase-by-Phase Approach: Teach the course in a sequence that mirrors the structure of a compiler: start with lexical analysis, then syntax analysis, semantic analysis, and so on. Each topic builds on the previous one, allowing students to see the progression from source code to executable code.
- Scaffold Learning: Begin with simple examples and gradually introduce more complexity. For instance, start with simple regular expressions before moving on to more complex lexical analysis tasks.

3. Hands-on Programming Exercises

- Implement Components: Assign programming exercises where students build individual components of a compiler, such as a lexer or parser. This helps them understand each phase in depth.
- Incremental Project Work: Have students start with a basic project (e.g., a simple calculator or interpreter) and add features incrementally, such as adding a syntax tree, type checking, or optimization.
- Use Compiler Tools: Introduce tools like Lex and Yacc early in the course, and provide assignments that require students to use these tools to generate parts of their compiler.

4. Active Learning Techniques

- Classroom Discussions and Debates: Encourage discussions on different parsing techniques, optimization strategies, or the design choices in well-known compilers. Debates can help students understand the trade-offs involved in compiler design.
- Peer Instruction: Use peer instruction methods where students explain concepts to each other. This can be particularly effective for complex topics like syntax-directed translation or register allocation.

5. Visualization and Diagrams

- Draw Parse Trees and Automata: Use diagrams to explain concepts like parse trees, finite automata, and control flow graphs. Visual representations help students grasp abstract concepts more easily.
- Flowcharts and Graphs: Use flowcharts to represent the data and control flow within a compiler, and control flow graphs (CFGs) to

explain optimization techniques.

6. Interactive and Adaptive Teaching

- Use Interactive Tools: Incorporate interactive tools or simulators that allow students to experiment with different phases of a compiler. For example, tools that visualize lexical analysis or parsing can make these concepts more tangible.
- Adaptive Feedback: Provide continuous feedback on assignments and projects. Adapt your teaching based on the common areas where students struggle, offering additional resources or explanations as needed.

7. Case Studies and Real-world Examples

- Analyze Existing Compilers: Use case studies of well-known compilers (like GCC or LLVM) to show how theoretical concepts are applied in practice. This can help bridge the gap between theory and real-world application.
- Reverse Engineering: Assign projects where students analyze or modify existing open-source compilers to understand how they are built and maintained.

8. Collaborative Learning

- Group Projects: Encourage students to work in groups on larger projects, such as implementing a compiler for a simple language. Collaborative projects can help students learn from each other and tackle more complex tasks than they could individually.
- Code Reviews: Organize peer code review sessions where students critique each other's compiler implementations, focusing on both correctness and optimization.

9. Integrate Theory with Practice

- Theory-Driven Labs: Follow up theoretical lectures with lab sessions where students implement the concepts discussed. For example, after teaching about LR parsing, have students implement an LR parser in a lab session.
- Practical Examples: During lectures, regularly switch between theoretical explanations and practical examples, such as showing how a regular expression is converted into a finite automaton, and then having students implement this in code.

10. Continuous Assessment and Feedback

- Frequent Quizzes and Assignments: Use short quizzes and frequent assignments to keep students engaged and to reinforce key concepts.

	• Iterative Project Feedback: Provide iterative feedback on project milestones, helping students correct mistakes and improve their understanding before the final submission. **11. Guest Lectures and Industry Insights** • Industry Experts: Invite guest speakers from industry to talk about real-world compiler development challenges, such as those faced in optimizing for modern processors or in just-in-time compilation. • Research Insights: Introduce students to current research trends in compiler design, such as machine learning-based optimizations, to inspire further learning and exploration. **12. End-of-Course Capstone Project** • Comprehensive Project: Conclude the course with a capstone project where students design and implement a simple compiler for a small programming language, integrating all the concepts learned throughout the course. • Project Presentation: Have students present their compiler projects, explaining the design choices, challenges faced, and solutions implemented. This reinforces their learning and allows for peer-to-peer learning. By combining these teaching strategies, you can create a well-rounded Compiler course that equips students with both the theoretical understanding and practical skills needed to design and build compilers.

Student Workload (SWL)

الحمل الدراسي للطالب محسوب لـ ١٥ أسبوعا

Structured SWL (h/sem) الحمل الدراسي المنتظم للطالب خلال الفصل	79	**Structured SWL (h/w)** الحمل الدراسي المنتظم للطالب أسبوعيا	5
Unstructured SWL (h/sem) الحمل الدراسي غير المنتظم للطالب خلال الفصل	46	**Unstructured SWL (h/w)** الحمل الدراسي غير المنتظم للطالب أسبوعيا	3
Total SWL (h/sem) الحمل الدراسي الكلي للطالب خلال الفصل	**125**		

Module Evaluation

تقييم المادة الدراسية

	Time/Number	Weight (Marks)	Week Due	Relevant Learning Outcome

Formative assessment	**Quizzes**	2	10% (10)	6 and 10	LO #1, #2, #3, #4, #5
	Assignments	2	8% (8)	4 and 11	LO #1, #2, #3, #4, #5
	Projects / Lab.	1	14% (14)	Continuous	All
	Report	1	8% (8)		LO #1, #2, #3, #4, #5.#6
Summative assessment	**Midterm Exam**	2hr	10% (10)	8	LO #1, #2, #3, #4
	Final Exam	4hr	50% (50)	16	All
Total assessment			100% (100 Marks)		

Delivery Plan (Weekly Syllabus) المنهاج الاسبوعي النظري	
	Material Covered
Week 1	**Introduction to Translators:** Compiler, Interpreter Assembler, Linker and Loader.
Week 2	**Structure of Compiler:** Analysis Phases, Construction Phases, Example of Compilation.
Week 3	**Lexical Analyzer:** Role of the Lexical Analyzer, Tasks of Lexical Analyzer, Input Buffering, Symbol Table
Week 4	**Recognition of Tokens:** Token, Lexeme, Pattern Specifications of Tokens, Operations on Languages
Week 5	**Syntax Analysis (Parse):** Role of the Parser, Interaction of lexical analyzer with parser, Parse Tree
Week 6	**Context-Free Grammars (CFG):** Ambiguity in Instructions, Regular Expression vs. CFG, Elimination of Left Recursion
Week 7	**Midterm Exam**
Week 8	**Parsing Techniques:** Top-Down Parser, Bottom-Up Parser, Compute of FIRST Function
Week 9	**Top-Down Parsing:** Backtracking, Predictive Parser, Design Parsing Table
Week 10	**Bottom-Up Parsing:** Shift-Reduce Parser, Operator-Precedence Parser
Week 11	**Semantic analysis:** Attributes and their computation, Type Checking, Semantic Errors
Week 12	**Intermediate Code Generation:** Control Structures, Three-Address Code, Operations Within Expressions
Week 13	**Code Optimization:** Introduction and View of Optimization, Global Optimization, Local Optimization.
Week 14	**Code Generation**: Issues in the Design of a Code Generator, Object Program, Assembly Controls
Week 15	**Preparatory week before the final Exam**
Week 16	**Final exam**

Delivery Plan (Weekly Lab. Syllabus) المنهاج الاسبوعي للمختبر

	Material Covered
Week 1	**Programming Languages:** Types and Features of Programming language
Week 2	**Types of Errors in source program, Errors Handling, Regular Expression**
Week 3	**Classification of source program:** using Lexical Analyzer, Tables Management
Week 4	**Transition Diagrams (TD), Lexical Errors. Assignment (1)**
Week 5	**Abstract syntax Tree, Derivation, Syntax Errors**
Week 6	**Context-Free Grammars (CFG):** Elimination of Left Factoring, Assignment (2)
Week 7	**Midterm Exam**
Week 8	**Parsing Techniques:** Compute of FOLLOW Function
Week 9	**Top-Down Parsing:** LL(1) Grammar, Conditions of LL(1)
Week 10	**Bottom-Up Parsing:** Table Construction of OPP, Assignment (3
Week 11	**Semantic analysis:** Run-Time Errors, Assignment (4)
Week 12	**Intermediate Code Generation:** Control Structures, Three-Address Code, Operations Within Expressions
Week 13	**Code Generation**: Simple Register Allocation, Assignment (5)
Week 14	Preparatory week before the final Exam
Week 15	**Project Discussion**

Learning and Teaching Resources
مصادر التعلم والتدريس

	Text	Available in the Library?
Required Texts	***[1]: E Alfred V. Aho, RaviSethi and Jeffrey D. Ullman, "Compilers: Principles, techniques, & tools" (2nd edition), 2007.*** ***[2]: Torben Mogensen, "Introduction to Compiler Design," Springer-Verlag London, 2011.***	
Recommended Texts	[1]: Torben Mogensen, "Basics of Compiler Design" DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF COPENHAGEN, 2010 [2]: Adesh K. Pandey, "CONCEPTS OF COMPILER DESIGN" Second Edition:2011. [3]: Dilshad Hasan, "Compiler Design", Knowledge Book Distributors, 1st Edition:2010.	
Websites		

Grading Scheme مخطط الدرجات				
Group	**Grade**	التقدير	**Marks %**	**Definition**
Success Group (50 - 100)	**A** - Excellent	امتياز	90 – 100	Outstanding Performance
	B - Very Good	جيد جدا	80 – 89	Above average with some errors
	C - Good	جيد	70 – 79	Sound work with notable errors
	D - Satisfactory	متوسط	60 – 69	Fair but with major shortcomings
	E - Sufficient	مقبول	50 – 59	Work meets minimum criteria
Fail Group (0 – 49)	**FX** – Fail	راسب (قيد المعالجة)	(45-49)	More work required but credit awarded
	F – Fail	راسب	(0-44)	Considerable amount of work required

Note: Marks Decimal places above or below 0.5 will be rounded to the higher or lower full mark (for example a mark of 54.5 will be rounded to 55, whereas a mark of 54.4 will be rounded to 54. The University has a policy NOT to condone "near-pass fails" so the only adjustment to marks awarded by the original marker(s) will be the automatic rounding outlined above.