

**San Pablo Catholic University (UCSP)**  
**Undergraduate Program in**  
**Computer Science**  
**SILABO**



**CS342. Compilers (Mandatory)**

**1. General information**

1.1 School	:	Ciencia de la Computación
1.2 Course	:	CS342. Compilers
1.3 Semester	:	8 <sup>vo</sup> Semestre.
1.4 Prerequisites	:	CS341. Programming languages . (7 <sup>th</sup> Sem)
1.5 Type of course	:	Mandatory
1.6 Learning modality	:	Virtual
1.7 Horas	:	2 HT; 2 HP; 2 HL;
1.8 Credits	:	4

**2. Professors**

**Lecturer**

- Gina Lucia Muñoz Salas <glmunoz@ucsp.edu.pe>  
– MSc in Ciencia de la Computación, Universidad Católica San Pablo, Perú, 2019.

**3. Course foundation**

That the student knows and understands the concepts and fundamental principles of the theory of compilation to realize the construction of a compiler

**4. Summary**

1. Program Representation 2. Language Translation and Execution 3. Syntax Analysis 4. Compiler Semantic Analysis  
5. Code Generation

**5. Generales Goals**

- Know the basic techniques used during the process of intermediate generation, optimization and code generation.
- Learning to implement small compilers.

**6. Contribution to Outcomes**

This discipline contributes to the achievement of the following outcomes:

- a) An ability to apply knowledge of mathematics, science. (**Assessment**)
- b) An ability to design and conduct experiments, as well as to analyze and interpret data. (**Assessment**)
- j) Apply the mathematical basis, principles of algorithms and the theory of Computer Science in the modeling and design of computational systems in such a way as to demonstrate understanding of the equilibrium points involved in the chosen option. (**Assessment**)

**7. Content**

<b>UNIT 1: Program Representation (5)</b>	
<b>Competences: a,b</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators</li> <li>• Abstract syntax trees; contrast with concrete syntax</li> <li>• Data structures to represent code for execution, translation, or transmission</li> <li>• Just-in-time compilation and dynamic recompilation</li> <li>• Other common features of virtual machines, such as class loading, threads, and security.</li> </ul>	<ul style="list-style-type: none"> <li>• Explain how programs that process other programs treat the other programs as their input data [Familiarity]</li> <li>• Describe an abstract syntax tree for a small language [Familiarity]</li> <li>• Describe the benefits of having program representations other than strings of source code [Familiarity]</li> <li>• Write a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator [Familiarity]</li> <li>• Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers [Familiarity]</li> <li>• Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation [Familiarity]</li> <li>• Identify the services provided by modern language run-time systems [Familiarity]</li> </ul>
<b>Readings:</b> Louden (2004b)	

<b>UNIT 2: Language Translation and Execution (10)</b>	
<b>Competences: a,b,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Interpretation vs. compilation to native code vs. compilation to portable intermediate representation</li> <li>• Language translation pipeline: parsing, optional type-checking, translation, linking, execution <ul style="list-style-type: none"> <li>– Execution as native code or within a virtual machine</li> <li>– Alternatives like dynamic loading and dynamic (or “just-in-time”) code generation</li> </ul> </li> <li>• Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)</li> <li>• Run-time layout of memory: call-stack, heap, static data <ul style="list-style-type: none"> <li>– Implementing loops, recursion, and tail calls</li> </ul> </li> <li>• Memory management <ul style="list-style-type: none"> <li>– Manual memory management: allocating, de-allocating, and reusing heap memory</li> <li>– Automated memory management: garbage collection as an automated technique using the notion of reachability</li> </ul> </li> </ul>	<ul style="list-style-type: none"> <li>• Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs interpreter, run-time representation of data objects, etc) [Assessment]</li> <li>• Distinguish syntax and parsing from semantics and evaluation [Assessment]</li> <li>• Sketch a low-level run-time representation of core language constructs, such as objects or closures [Assessment]</li> <li>• Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model [Assessment]</li> <li>• Identify and fix memory leaks and dangling-pointer dereferences [Assessment]</li> <li>• Discuss the benefits and limitations of garbage collection, including the notion of reachability [Assessment]</li> </ul>
<b>Readings:</b> Aho et al. (2011), Louden (2004a), Appel (2002), Teufel and Schmidt (1998)	

<b>UNIT 3: Syntax Analysis (10)</b>	
<b>Competences: a,b,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Scanning (lexical analysis) using regular expressions</li> <li>• Parsing strategies including top-down (e.g., recursive descent, Earley parsing, or LL) and bottom-up (e.g., backtracking or LR) techniques; role of context-free grammars</li> <li>• Generating scanners and parsers from declarative specifications</li> </ul>	<ul style="list-style-type: none"> <li>• Use formal grammars to specify the syntax of languages [Assessment]</li> <li>• Use declarative tools to generate parsers and scanners [Assessment]</li> <li>• Identify key issues in syntax definitions: ambiguity, associativity, precedence [Assessment]</li> </ul>
<b>Readings:</b> Aho et al. (2011), Louden (2004a), Appel (2002), Teufel and Schmidt (1998)	

UNIT 4: Compiler Semantic Analysis (15)	
Competences: a,b,j	
Content	Generales Goals
<ul style="list-style-type: none"> <li>• High-level program representations such as abstract syntax trees</li> <li>• Scope and binding resolution</li> <li>• Type checking</li> <li>• Declarative specifications such as attribute grammars</li> </ul>	<ul style="list-style-type: none"> <li>• Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences [Assessment]</li> <li>• Describe semantic analyses using an attribute grammar [Assessment]</li> </ul>
<b>Readings:</b> Aho et al. (2011), Louden (2004a), Appel (2002), Teufel and Schmidt (1998)	

UNIT 5: Code Generation (20)	
Competences: a,b,j	
Content	Generales Goals
<ul style="list-style-type: none"> <li>• Procedure calls and method dispatching</li> <li>• Separate compilation; linking</li> <li>• Instruction selection</li> <li>• Instruction scheduling</li> <li>• Register allocation</li> <li>• Peephole optimization</li> </ul>	<ul style="list-style-type: none"> <li>• Identify all essential steps for automatically converting source code into assembly or other low-level languages [Assessment]</li> <li>• Generate the low-level code for calling functions/methods in modern languages [Assessment]</li> <li>• Discuss why separate compilation requires uniform calling conventions [Assessment]</li> <li>• Discuss why separate compilation limits optimization because of unknown effects of calls [Assessment]</li> <li>• Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization [Assessment]</li> </ul>
<b>Readings:</b> Aho et al. (2011), Louden (2004a), Appel (2002), Teufel and Schmidt (1998)	

8. Methodology
<p>El profesor del curso presentará clases teóricas de los temas señalados en el programa propiciando la intervención de los alumnos.</p> <p>El profesor del curso presentará demostraciones para fundamentar clases teóricas.</p> <p>El profesor y los alumnos realizarán prácticas</p> <p>Los alumnos deberán asistir a clase habiendo leído lo que el profesor va a presentar. De esta manera se facilitará la comprensión y los estudiantes estarán en mejores condiciones de hacer consultas en clase.</p>

9. Assessment
<p><b>Continuous Assessment 1</b> : 20 %</p> <p><b>Partial Exam</b> : 30 %</p> <p><b>Continuous Assessment 2</b> : 20 %</p> <p><b>Final exam</b> : 30 %</p>

## References

- Aho, Alfred et al. (2011). *Compilers Principles Techniques And Tools*. 2nd. ISBN:10-970-26-1133-4. Pearson.
- Appel, A. W. (2002). *Modern compiler implementation in Java*. 2.a edición. Cambridge University Press.
- Louden, Kenneth C. (2004a). *Compiler Construction: Principles and Practice*. Thomson.
- Louden, Kenneth C. (2004b). *Lenguajes de Programacion*. Thomson.
- Teufel, Bernard and Stephanie Schmidt (1998). *Fundamentos de Compiladores*. Addison Wesley Iberoamericana.