

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



**CS341. Programming languages (Mandatory)**

**1. General information**

1.1 School	:	Ciencia de la Computación
1.2 Course	:	CS341. Programming languages
1.3 Semester	:	7 <sup>mo</sup> Semestre.
1.4 Prerequisites	:	CS211. Computer Science Theory. (4 <sup>th</sup> Sem)
1.5 Type of course	:	Mandatory
1.6 Learning modality	:	Face to face
1.7 Horas	:	2 HT; 2 HP; 2 HL;
1.8 Credits	:	4

**2. Professors**

**3. Course foundation**

Los lenguajes de programación son el medio a través del cual los programadores describen con precisión los conceptos, formulan algoritmos y representan sus soluciones. Un científico de la computación trabajará con diferentes lenguajes, por separado o en conjunto. Los científicos de la computación deben entender los modelos de programación de los diferentes lenguajes, tomar decisiones de diseño basados en el lenguaje de programación y sus conceptos. El profesional a menudo necesitará aprender nuevos lenguajes y construcciones de programación y debe entender los fundamentos de como las características del lenguaje de programación están definidas, compuestas e implementadas. El uso eficaz de los lenguajes de programación y la apreciación de sus limitaciones, también requiere un conocimiento básico de traducción de lenguajes de programación y su análisis de ambientes estáticos y dinámicos, así como los componentes de tiempo de ejecución tales como la gestión de memoria, entre otros detalles de relevancia.

**4. Summary**

1. 2. Language Pragmatics 3. Type Systems 4. Object-Oriented Programming 5. Functional Programming 6. Event-Driven and Reactive Programming 7. Logic Programming

**5. Generales Goals**

- Capacitar a los estudiantes para entender los lenguajes de programación desde diferentes tipos de vista, según el modelo subyacente, los componentes fundamentales presentes en todo lenguaje de programación y como objetos formales dotados de una estructura y un significado según diversos enfoques.

**6. Contribution to Outcomes**

This discipline contributes to the achievement of the following outcomes:

- a) An ability to apply knowledge of mathematics, science. (**Usage**)
- b) An ability to design and conduct experiments, as well as to analyze and interpret data. (**Usage**)
- i) An ability to use the techniques, skills, and modern computing tools necessary for computing practice. (**Usage**)
- 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. (**Usage**)

## 7. Content

### UNIT 1: (18)

**Competences: a,b,i,j**

#### Content

- Historia de los Lenguajes de Programación
- Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
- Data structures to represent code for execution, translation, or transmission
- Estructura de un programa: Léxico, Sintáctico y Semántico
- BNF
- Interpretation vs. compilation to native code vs. compilation to portable intermediate representation [Familiarity]

#### Generales Goals

- Reconocer el desarrollo histórico de los lenguajes de programación. [Familiarity]
- Identificar los paradigmas que agrupan a la mayoría de lenguajes de programación existentes hoy en día. [Familiarity]
- Explain how programs that process other programs treat the other programs as their input data [Familiarity]
- Describe an abstract syntax tree for a small language [Familiarity]
- Write a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator [Usage]
- Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs interpreter, run-time representation of data objects, etc) [Familiarity]
- Reconocer como funciona un programa a nivel de computador. [Familiarity]

**Readings:** Sebesta (2012), Webber (2010)

### UNIT 2: Language Pragmatics (12)

**Competences: a,b,i,j**

#### Content

- Principles of language design such as orthogonality
- Evaluation order, precedence and associativity
- Eager vs. delayed evaluation
- Defining control and iteration constructs
- External calls and system libraries

#### Generales Goals

- Discuss the role of concepts such as orthogonality and well-chosen defaults in language design [Usage]
- Use crisp and objective criteria for evaluating language-design decisions [Usage]
- Give an example program whose result can differ under different rules for evaluation order, precedence, or associativity [Usage]
- Show uses of delayed evaluation, such as user-defined control abstractions [Familiarity]
- Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation [Familiarity]

**Readings:** Sebesta (2012), Webber (2010), Roy and Haridi (2004)

<b>UNIT 3: Type Systems (18)</b>	
<b>Competences: a,b,i,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types</li> <li>• Type checking</li> <li>• Type safety as preservation plus progress</li> <li>• Type inference</li> <li>• Static overloading</li> </ul>	<ul style="list-style-type: none"> <li>• Define a type system precisely and compositionally [Usage]</li> <li>• For various foundational type constructors, identify the values they describe and the invariants they enforce [Familiarity]</li> <li>• Precisely specify the invariants preserved by a sound type system [Familiarity]</li> <li>• Prove type safety for a simple language in terms of preservation and progress theorems [Usage]</li> <li>• Implement a unification-based type-inference algorithm for a simple language [Usage]</li> <li>• Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs [Familiarity]</li> </ul>
<b>Readings:</b> Sebesta (2012), Webber (2010), Roy and Haridi (2004)	

UNIT 4: Object-Oriented Programming (12)	
Competences: a,b,i,j	
Content	Generales Goals
<ul style="list-style-type: none"> <li>• Object-oriented design <ul style="list-style-type: none"> <li>– Decomposition into objects carrying state and having behavior</li> <li>– Class-hierarchy design for modeling</li> </ul> </li> <li>• Definition of classes: fields, methods, and constructors</li> <li>• Subclasses, inheritance, and method overriding</li> <li>• Dynamic dispatch: definition of method-call</li> <li>• Subtyping <ul style="list-style-type: none"> <li>– Subtype polymorphism; implicit upcasts in typed languages</li> <li>– Notion of behavioral replacement: subtypes acting like supertypes</li> <li>– Relationship between subtyping and inheritance</li> </ul> </li> <li>• Object-oriented idioms for encapsulation <ul style="list-style-type: none"> <li>– Privacy and visibility of class members</li> <li>– Interfaces revealing only method signatures</li> <li>– Abstract base classes</li> </ul> </li> <li>• Using collection classes, iterators, and other common library components</li> </ul>	<ul style="list-style-type: none"> <li>• Design and implement a class [Usage]</li> <li>• Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses [Usage]</li> <li>• Correctly reason about control flow in a program using dynamic dispatch [Usage]</li> <li>• Compare and contrast (1) the procedural/functional approach—defining a function for each operation with the function body providing a case for each data variant—and (2) the object-oriented approach—defining a class for each data variant with the class definition providing a method for each operation Understand both as defining a matrix of operations and variants [Assessment]</li> <li>• Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype) [Usage]</li> <li>• Use object-oriented encapsulation mechanisms such as interfaces and private members [Usage]</li> <li>• Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language [Usage]</li> </ul>
<b>Readings:</b> Sebesta (2012), Webber (2010), Roy and Haridi (2004)	

<b>UNIT 5: Functional Programming (18)</b>	
<b>Competences: a,b,i,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Effect-free programming <ul style="list-style-type: none"> <li>– Function calls have no side effects, facilitating compositional reasoning</li> <li>– Variables are immutable, preventing unexpected changes to program data by other code</li> <li>– Data can be freely aliased or copied without introducing unintended effects from mutation</li> </ul> </li> <li>• Processing structured data (e.g., trees) via functions with cases for each data variant <ul style="list-style-type: none"> <li>– Associated language constructs such as discriminated unions and pattern-matching over them</li> <li>– Functions defined over compound data in terms of functions applied to the constituent pieces</li> </ul> </li> <li>• First-class functions (taking, returning, and storing functions)</li> <li>• Function closures (functions using variables in the enclosing lexical environment) <ul style="list-style-type: none"> <li>– Basic meaning and definition – creating closures at run-time by capturing the environment</li> <li>– Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments</li> <li>– Using a closure to encapsulate data in its environment</li> <li>– Currying and partial application</li> </ul> </li> <li>• Defining higher-order operations on aggregates, especially map, reduce/fold, and filter</li> </ul>	<ul style="list-style-type: none"> <li>• Write basic algorithms that avoid assigning to mutable state or considering reference equality [Usage]</li> <li>• Write useful functions that take and return other functions [Usage]</li> <li>• Compare and contrast (1) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and (2) the object-oriented approach-defining a class for each data variant with the class definition providing a method for each operation Understand both as defining a matrix of operations and variants [Assessment]</li> <li>• Correctly reason about variables and lexical scope in a program using function closures [Usage]</li> <li>• Use functional encapsulation mechanisms such as closures and modular interfaces [Usage]</li> <li>• Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language [Usage]</li> </ul>
<b>Readings: Sebesta (2012), Webber (2010), Roy and Haridi (2004)</b>	

<b>UNIT 6: Event-Driven and Reactive Programming (12)</b>	
<b>Competences: a,b,i,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Events and event handlers</li> <li>• Canonical uses such as GUIs, mobile devices, robots, servers</li> <li>• Using a reactive framework <ul style="list-style-type: none"> <li>– Defining event handlers/listeners</li> <li>– Main event loop not under event-handler-writer's control</li> </ul> </li> <li>• Externally-generated events and program-generated events</li> <li>• Separation of model, view, and controller</li> </ul>	<ul style="list-style-type: none"> <li>• Write event handlers for use in reactive systems, such as GUIs [Usage]</li> <li>• Explain why an event-driven programming style is natural in domains where programs react to external events [Familiarity]</li> <li>• Describe an interactive system in terms of a model, a view, and a controller [Familiarity]</li> </ul>
<b>Readings:</b> Sebesta (2012)	

<b>UNIT 7: Logic Programming (12)</b>	
<b>Competences: a,b,i,j</b>	
<b>Content</b>	<b>Generales Goals</b>
<ul style="list-style-type: none"> <li>• Causal representation of data structures and algorithms</li> <li>• Unification</li> <li>• Backtracking and search</li> <li>• Cuts</li> </ul>	<ul style="list-style-type: none"> <li>• Use a logic language to implement a conventional algorithm [Usage]</li> <li>• Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts [Usage]</li> </ul>
<b>Readings:</b> Sebesta (2012), Webber (2010), Roy and Haridi (2004)	

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

- Roy, Peter Van and Seif Haridi (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press: Cambridge, MA, USA. ISBN: 0262220695.
- Sebesta, Robert W. (2012). *Concepts of Programming Languages*. 10th. Addison-Wesley Publishing Company: USA. ISBN: 0131395319.
- Webber, Adam Brooks (2010). *Modern Programming Languages: A Practical Introduction*. 2nd. Franklin, Beedle and Associates, Inc. ISBN: 978-1-59028-250-2.