Programming Languages (PL)

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. In the course of a career, a computer scientist will work with many different languages, separately or together. Software developers must understand the programming models underlying different languages and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs, and must understand the principles underlying how programming language features are defined, composed, and implemented. The effective use of programming languages, and appreciation of their limitations, also requires a basic knowledge of programming language translation and static program analysis, as well as run-time components such as memory management.
### PL. Programming Languages (8 Core-Tier1 hours, 20 Core-Tier2 hours)

<table>
<thead>
<tr>
<th></th>
<th>Core-Tier1 hours</th>
<th>Core-Tier2 hours</th>
<th>Includes Electives</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/Object-Oriented Programming</td>
<td>4</td>
<td>6</td>
<td>N</td>
</tr>
<tr>
<td>PL/Functional Programming</td>
<td>3</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>PL/Event-Driven and Reactive Programming</td>
<td>2</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>PL/Basic Type Systems</td>
<td>1</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>PL/Program Representation</td>
<td>1</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>PL/Language Translation and Execution</td>
<td>3</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>PL/Syntax Analysis</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Compiler Semantic Analysis</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Code Generation</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Runtime Systems</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Static Analysis</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Advanced Programming Constructs</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Concurrency and Parallelism</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Type Systems</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Formal Semantics</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Language Pragmatics</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PL/Logic Programming</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

**Note:**
- Some topics from one or more of the first three Knowledge Units (Object-Oriented Programming, Functional Programming, Event-Driven and Reactive Programming) are likely to be integrated with topics in the SF-Software Development Fundamentals Knowledge Area in a curriculum’s introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- Some of the most important core learning outcomes are relevant to object-oriented programming, functional programming, and, in fact, all programming. These learning outcomes are repeated in the Object-Oriented Programming and Functional Programming Knowledge Units, with a note to this effect. We do not intend that a
curriculum necessarily needs to cover them multiple times, though some will. We repeat them only because they do not naturally fit in only one Knowledge Unit.

PL/Object-Oriented Programming

[4 Core-Tier1 hours, 6 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Object-oriented design
  - Decomposition into objects carrying state and having behavior
  - Class-hierarchy design for modeling
- Definition of classes: fields, methods, and constructors
- Subclasses, inheritance, and method overriding
- Dynamic dispatch: definition of method-call

[Core-Tier2]

- Subtyping (cross-reference PL/Type Systems)
  - Subtype polymorphism; implicit upcasts in typed languages
  - Notion of behavioral replacement: subtypes acting like supertypes
  - Relationship between subtyping and inheritance
- Object-oriented idioms for encapsulation
  - Privacy and visibility of class members
  - Interfaces revealing only method signatures
  - Abstract base classes
- Using collection classes, iterators, and other common library components

Learning outcomes:

[Core-Tier1]

1. Design and implement a class. [Usage]
2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Usage]
3. Correctly reason about control flow in a program using dynamic dispatch. [Usage]
4. 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] This outcome also appears in PL/Functional Programming.

[Core-Tier2]

5. 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). [Familiarity]
6. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Usage]
7. 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] This outcome also appears in PL/Functional Programming.
PL/Functional Programming

[3 Core-Tier1 hours, 4 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Effect-free programming
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
  - Data can be freely aliased or copied without introducing unintended effects from mutation
- Processing structured data (e.g., trees) via functions with cases for each data variant
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Functions defined over compound data in terms of functions applied to the constituent pieces
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- Function closures (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition -- creating closures at run-time by capturing the environment
  - Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
  - Currying and partial application
- Defining higher-order operations on aggregates, especially map, reduce/fold, and filter

Learning outcomes:

[Core-Tier1]

1. Write basic algorithms that avoid assigning to mutable state or considering reference equality. [Usage]
2. Write useful functions that take and return other functions. [Usage]
3. 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] This outcome also appears in PL/Object-Oriented Programming.

[Core-Tier2]

4. Correctly reason about variables and lexical scope in a program using function closures. [Usage]
5. Use functional encapsulation mechanisms such as closures and modular interfaces. [Usage]
6. 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] This outcome also appears in PL/Object-Oriented Programming.
PL/Event-Driven and Reactive Programming

[2 Core-Tier2 hours]

This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.

Topics:

- Events and event handlers
- Canonical uses such as GUIs, mobile devices, robots, servers
- Using a reactive framework
  - Defining event handlers/listeners
  - Main event loop not under event-handler-writer’s control
- Externally-generated events and program-generated events
- Separation of model, view, and controller

Learning outcomes:

1. Write event handlers for use in reactive systems, such as GUIs. [Usage]
2. Explain why an event-driven programming style is natural in domains where programs react to external events. [Familiarity]
3. Describe an interactive system in terms of a model, a view, and a controller. [Familiarity]

PL/Basic Type Systems

[1 Core-Tier1 hour, 4 Core-Tier2 hours]

The core-tier2 hours would be profitably spent both on the core-tier2 topics and on a less shallow treatment of the core-tier1 topics and learning outcomes.

Topics:

[Core-Tier1]

- A type as a set of values together with a set of operations
  - Primitive types (e.g., numbers, Booleans)
  - Compound types built from other types (e.g., records, unions, arrays, lists, functions, references)
- Association of types to variables, arguments, results, and fields
- Type safety and errors caused by using values inconsistently given their intended types
- Goals and limitations of static typing
  - Eliminating some classes of errors without running the program
  - Undecidability means static analysis must conservatively approximate program behavior

[Core-Tier2]

- Generic types (parametric polymorphism)
  - Definition
  - Use for generic libraries such as collections
  - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
- Complementary benefits of static and dynamic typing
  - Errors early vs. errors late/avoided
Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections

Avoid misuse of code vs. allow more code reuse

Detect incomplete programs vs. allow incomplete programs to run

**Learning outcomes:**

[Core-Tier1]
1. For both a primitive and a compound type, informally describe the values that have that type. [Familiarity]
2. For a language with a static type system, describe the operations that are forbidden statically, such as passing the wrong type of value to a function or method. [Familiarity]
3. Describe examples of program errors detected by a type system. [Familiarity]
4. For multiple programming languages, identify program properties checked statically and program properties checked dynamically. [Usage]
5. Give an example program that does not type-check in a particular language and yet would have no error if run. [Familiarity]
6. Use types and type-error messages to write and debug programs. [Usage]

[Core-Tier2]
7. Explain how typing rules define the set of operations that are legal for a type. [Familiarity]
8. Write down the type rules governing the use of a particular compound type. [Usage]
9. Explain why undecidability requires type systems to conservatively approximate program behavior. [Familiarity]
10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections. [Usage]
11. Discuss the differences among generics, subtyping, and overloading. [Familiarity]
12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software. [Familiarity]

**PL/Program Representation**

[1 Core-Tier2 hour]

**Topics:**

- Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
- Abstract syntax trees; contrast with concrete syntax
- Data structures to represent code for execution, translation, or transmission

**Learning outcomes:**

1. Explain how programs that process other programs treat the other programs as their input data. [Familiarity]
2. Describe an abstract syntax tree for a small language. [Usage]
3. Describe the benefits of having program representations other than strings of source code. [Familiarity]
4. Write a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator. [Usage]
PL/Language Translation and Execution

[3 Core-Tier2 hours]

Topics:

- Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
- Language translation pipeline: parsing, optional type-checking, translation, linking, execution
  - Execution as native code or within a virtual machine
  - Alternatives like dynamic loading and dynamic (or “just-in-time”) code generation
- Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
- Run-time layout of memory: call-stack, heap, static data
  - Implementing loops, recursion, and tail calls
- Memory management
  - Manual memory management: allocating, de-allocating, and reusing heap memory
  - Automated memory management: garbage collection as an automated technique using the notion of reachability

Learning outcomes:

1. Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Familiarity]
2. Distinguish syntax and parsing from semantics and evaluation. [Familiarity]
3. Sketch a low-level run-time representation of core language constructs, such as objects or closures. [Familiarity]
4. 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. [Familiarity]
5. Identify and fix memory leaks and dangling-pointer dereferences. [Usage]
6. Discuss the benefits and limitations of garbage collection, including the notion of reachability. [Familiarity]

PL/Syntax Analysis

[Elective]

Topics:

- Scanning (lexical analysis) using regular expressions
- 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
- Generating scanners and parsers from declarative specifications

Learning outcomes:

1. Use formal grammars to specify the syntax of languages. [Usage]
2. Use declarative tools to generate parsers and scanners. [Usage]
3. Identify key issues in syntax definitions: ambiguity, associativity, precedence. [Familiarity]
PL/Compiler Semantic Analysis

[Elective]

Topics:

- High-level program representations such as abstract syntax trees
- Scope and binding resolution
- Type checking
- Declarative specifications such as attribute grammars

Learning outcomes:

1. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Usage]
2. Describe semantic analyses using an attribute grammar. [Usage]

PL/Code Generation

[Elective]

Topics:

- Procedure calls and method dispatching
- Separate compilation; linking
- Instruction selection
- Instruction scheduling
- Register allocation
- Peephole optimization

Learning outcomes:

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Familiarity]
2. Generate the low-level code for calling functions/methods in modern languages. [Usage]
3. Discuss why separate compilation requires uniform calling conventions. [Familiarity]
4. Discuss why separate compilation limits optimization because of unknown effects of calls. [Familiarity]
5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization. [Familiarity]

PL/Runtime Systems

[Elective]

Topics:

- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
Data layout for objects and activation records
Just-in-time compilation and dynamic recompilation
Other common features of virtual machines, such as class loading, threads, and security.

Learning outcomes:
1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Familiarity]
2. Discuss benefits and limitations of automatic memory management. [Familiarity]
3. 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]
4. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation. [Familiarity]
5. Identify the services provided by modern language run-time systems. [Familiarity]

PL/Static Analysis
[Elective]
Topics:
- Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment
- Undecidability and consequences for program analysis
- Flow-insensitive analyses, such as type-checking and scalable pointer and alias analyses
- Flow-sensitive analyses, such as forward and backward dataflow analyses
- Path-sensitive analyses, such as software model checking
- Tools and frameworks for defining analyses
- Role of static analysis in program optimization
- Role of static analysis in (partial) verification and bug-finding

Learning outcomes:
1. Define useful static analyses in terms of a conceptual framework such as dataflow analysis. [Usage]
2. Explain why non-trivial sound static analyses must be approximate. [Familiarity]
3. Communicate why an analysis is correct (sound and terminating). [Usage]
4. Distinguish “may” and “must” analyses. [Familiarity]
5. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Familiarity]
6. Use the results of a static analysis for program optimization and/or partial program correctness. [Usage]

PL/Advanced Programming Constructs
[Elective]
Topics:
- Lazy evaluation and infinite streams
- Control Abstractions: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
• Metaprogramming: Macros, Generative programming, Model-based development
• Module systems
• String manipulation via pattern-matching (regular expressions)
• Dynamic code evaluation (“eval”)
• Language support for checking assertions, invariants, and pre/post-conditions

Learning outcomes:

1. Use various advanced programming constructs and idioms correctly. [Usage]
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Familiarity]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

PL/Concurrency and Parallelism

[Elective]

Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this elective Knowledge Unit aims only to complement the material included elsewhere in the Body of Knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.

Cross-reference PD/Parallel and Distributed Computing, SF/Parallelism.

Topics:

• Constructs for thread-shared variables and shared-memory synchronization
• Actor models
• Futures
• Language support for data parallelism
• Models for passing messages between sequential processes
• Effect of memory-consistency models on language semantics and correct code generation

Learning outcomes:

1. Write correct concurrent programs using multiple programming models, such as shared memory, actors, futures, and data-parallelism primitives. [Usage]
2. Use a message-passing model to analyze a communication protocol. [Usage]
3. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Familiarity]
PL/Type Systems

[Elective]

Topics:

- Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
- Type checking
- Type safety as preservation plus progress
- Type inference
- Static overloading

Learning outcomes:

1. Define a type system precisely and compositionally. [Usage]
2. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Familiarity]
3. Precisely specify the invariants preserved by a sound type system. [Familiarity]
4. Prove type safety for a simple language in terms of preservation and progress theorems. [Usage]
5. Implement a unification-based type-inference algorithm for a simple language. [Usage]
6. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs. [Familiarity]

PL/Formal Semantics

[Elective]

Topics:

- Syntax vs. semantics
- Lambda Calculus
- Approaches to semantics: Operational, Denotational, Axiomatic
- Proofs by induction over language semantics
- Formal definitions and proofs for type systems (cross-reference PL/Type Systems)
- Parametricity (cross-reference PL/Type Systems)
- Using formal semantics for systems modeling

Learning outcomes:

1. Give a formal semantics for a small language. [Usage]
2. Write a lambda-calculus program and show its evaluation to a normal form. [Usage]
3. Discuss the different approaches of operational, denotational, and axiomatic semantics. [Familiarity]
4. Use induction to prove properties of all programs in a language. [Usage]
5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system. [Usage]
6. Use parametricity to establish the behavior of code given only its type. [Usage]
7. Use formal semantics to build a formal model of a software system other than a programming language. [Usage]
PL/Language Pragmatics

[Elective]

Topics:
- 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

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

PL/Logic Programming

[Elective]

Topics:
- Clausal representation of data structures and algorithms
- Unification
- Backtracking and search
- Cuts

Learning outcomes:
1. Use a logic language to implement a conventional algorithm. [Usage]
2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts. [Usage]