Programming Languages

Preamble

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. Over 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 program analysis, as well as run-time components such as memory management.

Changes since the 2013 ACM/IEEE-CS Computer Science curricular guidelines (CS2013) include a redistribution of content formerly identified as core tier-1 and core tier-2 within the Programming Language Knowledge Area (KA). These are now CS Core hours and KA Core hours. All computer science graduates are expected to have the CS Core hours, and those graduates that specialize in a knowledge area are also expected to have the KA core hours. Content that is not identified as either CS Core hours or KA Core hours are elective topics. Specifically, the changes are:

- Object-Oriented Programming -2 KA Core hours
- Event-Driven and Reactive Programming +1 CS Core hours
- Basic Type Systems +2 CS Core hours, -2 KA Core hours
- Advanced Programming Concepts +1 CS Core hours
- Distribution, Concurrency and Parallelism +1 CS Core hours

In addition, some knowledge units are renamed to more accurately reflect their content:
- Static Analysis is renamed to Program Analyzers
- Concurrency and Parallelism is renamed to Distribution, Concurrency and Parallelism

Two new knowledge units have been added to reflect their growing importance as we look toward the 2030s:
- Formal Development Methodologies
- Hardware Interface

Allocation of Core Hours

PL. Programming Languages (13 CS Core hours, 16 KA Core hours)

<table>
<thead>
<tr>
<th></th>
<th>CS Core hours</th>
<th>KA Core hours</th>
<th>Includes Electives</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/Object-Oriented Programming</td>
<td>4</td>
<td>4</td>
<td>N</td>
</tr>
<tr>
<td>PL/Functional Programming</td>
<td>3</td>
<td>4</td>
<td>N</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 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.
- Some topics from the Distribution, Concurrency and Parallelism Knowledge Unit are likely to be integrated within the curriculum with topics from the Parallel and Distributed Programming Knowledge Area.
- Some topics from the Hardware Interface Knowledge Unit are likely to be integrated within the curriculum with topics from the System Fundamentals Knowledge Area.

Description of Knowledge Units

**PL/Object-Oriented Programming**

[4 CS Core hours, 4 KA Core hours]

- Topics
  - [CS Core]
    - 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
Exception handling

[KA Core]
Subtyping (also covered in 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

Illustrative learning outcomes
[CS Core]
1. Compose a class through design, implementation, and testing to meet behavioral requirements. [Creating]
2. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses. [Applying]
3. Predict and validate control flow in a program using dynamic dispatch. [Analyzing, Evaluating]
4. Compare and contrast
   a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and
   b) 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. [Analyzing]
This outcome also appears in PL/Functional Programming

[KA Core]
1. 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). [Understanding]
2. Use object-oriented encapsulation mechanisms such as interfaces and private members. [Applying]
3. 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. [Enumerate, Applying]
This outcome also appears in PL/Functional Programming.
PL/Functional Programming
[3 CS Core hours, 4 KA Core hours]

- Topics
  - [CS Core]
    - 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)
  - [KA Core]
    - 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
Illustrative learning outcomes

[CS Core]
1. Develop basic algorithms that avoid assigning to mutable state or considering reference equality. [Creating]
2. Develop useful functions that take and return other functions. [Creating]
3. Compare and contrast
   a) the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant, and
   b) 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. [Analyzing]
This outcome also appears in PL/Object-Oriented Programming.

[KA Core]
4. Correctly interpret variables and lexical scope in a program using function closures. [Understanding]
5. Use functional encapsulation mechanisms such as closures and modular interfaces. [Applying]
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. [Enumerate, Applying]
This outcome also appears in PL/Object-Oriented Programming.

PL/Event-Driven and Reactive Programming
[1 CS Core hours, 2 KA Core 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

[CS Core]
- Events and event handlers
- Canonical uses such as GUIs, mobile devices, robots, servers

[KA Core]
- 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

Illustrative Learning Outcomes

[CS Core]
1. Implement event handlers for use in reactive systems, such as GUIs. [Applying]
2. Examine why an event-driven programming style is natural in domains where programs react to external events. [Analyzing]

- [KA Core]

3. Describe an interactive system in terms of a model, a view, and a controller. [Understanding]

**PL/Basic Type Systems**

[3 CS Core hours, 2 KA Core hours]

The KA Core hours would be profitably spent both on the KA Core topics and on a less shallow treatment of the CS Core topics and learning outcomes.

- **Topics**
  - [CS Core]
    - 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
    - Generic types (parametric polymorphism)
      - Definition
      - Generic parameters and typing
      - Use for generic libraries such as collections
      - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
  - [KA Core]
    - 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
• Illustrative Learning Outcomes
  • [CS Core]
    1. Describe, for both a primitive and a compound type, the values that have that type. [Understanding]
    2. Describe, for a language with a static type system, the operations that are forbidden statically, such as passing the wrong type of value to a function or method. [Understanding]
    3. Describe examples of program errors detected by a type system. [Understanding]
    4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically. [Enumerate]
    5. Describe an example program that does not type-check in a particular language and yet would have no error if run. [Understanding]
    6. Use types and type-error messages to write and debug programs. [Applying]
  • [KA Core]
    7. Explain how typing rules define the set of operations that are legal for a type. [Understanding]
    8. List the type rules governing the use of a particular compound type. [Enumerate]
    9. Explain why undecidability requires type systems to conservatively approximate program behavior. [Understanding]
   10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections. [Enumerate]
   11. Discuss the differences among generics, subtyping, and overloading. [Understanding]
   12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software. [Understanding]

PL/Program Representation
[1 KA Core 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
  • Binary representation

• Illustrative Learning Outcomes
  1. Explain how programs that process other programs treat the other programs as their input data. [Understanding]
  2. Describe an abstract syntax tree for a small language. [Understanding]
3. Describe the benefits of having program representations other than strings of source code. [Understanding]

4. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator. [Applying]

**PL/Language Translation and Execution**

[3 KA Core 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
    - Use of sophisticated (complex) type systems, e.g., Rust.

- **Illustrative Learning Outcomes**
  1. Differentiate a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Understanding]
  2. Differentiate syntax and parsing from semantics and evaluation. [Understanding]
  3. Diagram a low-level run-time representation of core language constructs, such as objects or closures. [Applying]
  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. [Understanding]
  5. Investigate, identify and fix memory leaks and dangling-pointer dereferences. [Applying]
  6. Discuss the benefits and limitations of garbage collection, including the notion of reachability. [Understanding]
PL/Syntax Analysis
[Elective]
- **Topics**
  - Scanning (lexical analysis)
    - Using regular expressions
    - BNF
  - 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
  - Language theory
    - Chomsky hierarchy
    - Left-most/right-most derivation and ambiguity
    - Grammar transformation
  - Error recovery mechanisms
  - Generating scanners and parsers from declarative specifications

- **Illustrative Learning Outcomes**
  - Use formal grammars to specify the syntax of languages. [Applying]
  - Use declarative tools to generate parsers and scanners. [Applying]
  - Recognize key issues in syntax definitions: ambiguity, associativity, precedence. [Enumerate]

PL/Compiler Semantic Analysis
[Elective]
- **Topics**
  - High-level program representations such as abstract syntax trees
  - Scope and binding resolution
  - Type checking
    - Structural vs name equivalence
  - L-values/R-values
  - Call semantics
  - Parameter passing mechanisms
  - Declarative specifications such as attribute grammars

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

PL/Code Generation
[Elective]
- **Topics**
  - Procedure calls and method dispatching
● Separate compilation; linking
● Instruction selection
● Instruction scheduling
● Register allocation
● Peephole optimization

● Illustrative Learning Outcomes
  1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Enumerate]
  2. Generate the low-level code for calling functions/methods in modern languages. [Creating]
  3. Discuss why separate compilation requires uniform calling conventions. [Understanding]
  4. Discuss why separate compilation limits optimization because of unknown effects of calls. [Understanding]
  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. [Understanding]

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
  ● Process models
  ● Other common features of virtual machines, such as class loading, threads, and security.

● Illustrative Learning Outcomes
  1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Analyzing]
  2. Discuss benefits and limitations of automatic memory management. [Understanding]
  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. [Understanding]
  4. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation. [Understanding]
  5. Identify the services provided by modern language run-time systems. [Enumerate]
PL/Program Analyzers
[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 program analysis in (partial) verification and bug-finding

- Illustrative Learning Outcomes
  1. Define useful program analyses in terms of a conceptual framework such as dataflow analysis. [Enumerate]
  2. Explain why non-trivial sound program analyses must be approximate. [Understanding]
  3. Argue why an analysis is correct (sound and terminating). [Analyzing]
  4. Distinguish "may" and "must" analyses. [Familiarity]
  5. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Understanding]
  6. Use the results of a program analysis for program optimization and/or partial program correctness. [Applying]

PL/Advanced Programming Constructs
[1 CS Core hour, Elective]
- Topics
  - [CS Core]
    - Lazy evaluation and infinite streams
    - Control Abstractions: Exception Handling, Continuations, Monads
    - Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
  - [Elective]
    - 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
    - Domain specific languages, such as database languages, data science languages
Illustrative Learning Outcomes

[Core-Tier1]
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]

[Elective]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Familiarity]

PL/Distribution, Concurrency and Parallelism
[1 CS Core hour, 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

[CS Core]
- Constructs for thread-shared variables and shared-memory synchronization
- Actor models
- Asynchronous programming
- Models for passing messages between sequential processes

[Elective]
- Futures
- Language support for data parallelism
- Effect of memory-consistency models on language semantics and correct code generation
- Loci of control
- Representational State Transfer Application Programming Interfaces (REST APIs)
- Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing

Illustrative Learning Outcomes

[CS Core]
1. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, and data-parallelism primitives. [Applying]
2. Use a message-passing model to analyze a communication protocol. [Applying]
● [Elective]
  3. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Understanding]
  4. Explain how REST API's integrate applications and automate processes [Understanding].
  5. Explain benefits, constraints and challenges related to distributed and parallel computing [Understanding].

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

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

PL/Logic Programming
[Elective]
● Topics
  ● Clausal representation of data structures and algorithms
  ● Unification
  ● Backtracking and search
  ● Cuts

● Illustrative Learning Outcomes
  1. Use a logic language to implement a conventional algorithm. [Applying]
  2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts. [Applying]
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
  - Subtyping (cross-reference PL/Type Systems) [also covered in PL/Object-Oriented Programming]
    - Subtype polymorphism; implicit upcasts in typed languages
    - Notion of behavioral replacement: subtypes acting like supertypes
    - Relationship between subtyping and inheritance
  - Type safety as preservation plus progress
  - Type inference
  - Static overloading

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

PL/Formal Semantics
[Elective]

- Topics
  - Syntax vs. semantics
  - Lambda Calculus
  - Constraint checkers
  - Symbolic execution
  - 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

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

**PL/Formal Development Methodologies**

[Elective]

- **Topics**
  - Formal specification languages and methodologies
  - Theorem provers and logics
  - Specification and proof discharge for fully verified software systems (e.g. dafny, Coq, F*)
  - Formal modelling and manual refinement/implemention of software systems (e.g. TLA+)
  - Development with checker assisted DSLs and specialized languages (e.g. Liquid Haskell and Ivy)
  - Use of symbolic testing and symbolic model checking systems in the overall software development system.
  - Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

- **Illustrative Learning Outcomes**
  1. Use formal modeling techniques to develop and validate architectures. [Applying]
  2. Use proof assisted programming languages to develop fully specified and verified software artifacts. [Applying]
  3. Use verifier and specification support in programming languages to formally validate system properties. [Applying]
  4. Integrate symbolic validation tooling into a programming workflow. [Analyzing]
  5. Discuss when and how formal methods can be effectively used in the development process. [Understanding]

**PL/Hardware Interface**

[Elective]

- **Topics**
  - Embedded Systems
    - Microcontrollers
Interrupts and feedback
- Sensors/actuators
- GPU technology
- Field Programmable Gate Arrays
- Energy efficiency

Real-time systems
- Hard real-time systems vs soft real-time systems
- Timeliness
- Time synchronization/scheduling
- Prioritization
- Latency
- Compute jitter

Memory management (cross-reference: PL/Language Translation and Execution)
- Mapping programming construct (variable) to a memory location
- Shared memory
- Manual memory management
- Garbage collection

Illustrative Learning Outcomes
1. Design and develop software to interact with and control hardware [Creating]
2. Design methods for real-time systems [Creating]
3. Evaluate real-time scheduling and schedulability analysis [Analyzing]
4. Evaluate formal specification and verification of timing constraints and properties [Analyzing]

List of Professional Dispositions Appropriate for this KA
- Professional
- Inventive
- Meticulous
- Responsible

Math needed and wanted
- Discrete Mathematics
- Logic

Crosscutting and Overlapping topics
- PL/Distribution, Concurrency and Parallelism overlaps with PD/Parallel and Distributed Computing, SF/Parallelism

Subcommittee
Chair: Michael Oudshoorn, High Point University, NC, USA
Subcommittee members

- Annette Bieniusa, TU Kaiserslautern, Germany
- Alan Dearle, University of St. Andrews, Scotland
- Michelle Kuttel, University of Cape Town, South Africa
- Doug Lea, State University of New York at Oswego, NY, USA
- James Noble, Victoria University of Wellington, New Zealand
- Mark Marron, Microsoft Research, WA, USA
- Peter-Michael Osera, Grinnell College, IA, USA
- Michelle Mills Strout, University of Arizona, AZ, USA

Other contributors:

-