Foundations of Programming Languages (FPL)

Preamble

This knowledge area provides a basis (rooted in discrete mathematics and logic) for the understanding of complex modern programming languages: their foundations, implementation, and formal description. Although programming languages vary according to the language paradigm and the problem domain and evolve in response to both societal needs and technological advancement, they share an underlying abstract model of computation and program development. This remains true even as processor hardware and their interface with programming tools are becoming increasingly intertwined and progressively more complex. An understanding of the common abstractions and programming paradigms enables faster learning of new programming languages.

The Foundations of Programming Languages Knowledge Area is concerned with articulating the underlying concepts and principles of programming languages, the formal specification of a programming language and the behavior of a program, explaining how programming languages are implemented, comparing the strengths and weaknesses of various programming paradigms, and describing how programming languages interface with entities such as operating systems and hardware. The concepts covered in this area are applicable to many different languages and an understanding of these principles assists in being able to move readily from one language to the next, and to be able to select a programming paradigm and a programming language to best suit the problem at hand.

Two example courses are presented at the end of this knowledge area to illustrate how the content may be covered. The first is an introductory course which covers the CS Core and KA Core content. This is a course focused on the different programming paradigms and ensue familiarity with each to a level sufficient to be able to decide which paradigm is appropriate in which circumstances.

The second course is an advanced course focused on the implementation of a programming language and the formal description of a programming language and a formal description of the behavior of a program.

While these two courses have been the predominant way to cover this knowledge area of the past decade, it is by no means the only way that the content can be covered. An institution could, for example, choose to cover only the CS Core content (28 hours) in a shorter course, or in a course which combines this CS Core content with Core content from another knowledge area such as Software Engineering. Natural combinations are easily identifiable since they are the areas in which the Foundations of Programming

Languages knowledge areas overlaps with other knowledge areas. A list of such overlap areas is provided at the end of this knowledge area.

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 need to learn and work with many different languages, separately or together. Software developers must understand the programming models, new programming features and constructs, 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 to improve execution efficiency and long-term maintenance of developed software. The effective use of programming languages and appreciation of their limitations also requires a basic knowledge of programming language translation and program analysis, of run-time behavior, and components such as memory management and the interplay of concurrent processes communicating with each other through message-passing, shared memory, and synchronization, Finally, some developers and researchers will need to design new languages, an exercise which requires familiarity with basic principles.

Changes since CS 2013

These include a change in name of the Knowledge Area from Programming Languages to Foundations of Programming Languages to better reflect the fact that the KA is about the fundamentals underpinning programming languages and related concepts, and not about any specific programming languages. Changes also include a redistribution of content formerly identified as core tier-1 and core tier-2 within the Programming Language Knowledge Area (KA). In CS2013, graduates were expected to complete all tier-1 hours and 80% of the tier-2 hours, a total of 24 required hours. These are now CS Core hours. The remaining tier-2 hours are now 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 majority of the KA core hours. Content that is not identified as either CS Core hours or KA Core hours are non-core topics. The variation in core hours (tier-1 plus 80% of tier-2 hours) from 2013 reflects the change in importance or relevance of topics over the past decade. The inclusion of new topics is driven by their current prominence in the programming language landscape, or the anticipated impact of emerging areas on the profession in general. Specifically, the changes are:

● Object-Oriented Programming	-3 CS Core hours
● Functional Programming	-2 CS Core hours
● Event-Driven and Reactive Programming	+1 CS Core hour
● Parallel and Distributed Computing	+2 CS Core hours
● Type Systems	-1 CS Core hour
● Language Translation and Execution	-3 CS Core hours

In addition, some knowledge units from CS 2013 are renamed to more accurately reflect their content:

- Static Analysis is renamed to Program Analysis and Analyzers
- Concurrency and Parallelism is renamed to Parallel and Distributed Computing
- Program Representation is renamed to Program Abstraction and Representation
- Runtime Systems is renamed to Runtime Behavior and Systems
- Basic Type Systems and Type Systems were merged into a single topic and named Type Systems

Seven new knowledge units have been added to reflect their continuing and growing importance as we look toward the 2030s:

- Compiler vs Interpreted Languages +1 CS Core hour
- Scripting +2 CS Core hours
- Systems Execution and Memory Model +3 CS Core hours
- Formal Development Methodologies
- Design Principles of Programming Languages
- Quantum Computing
- Fundamentals of Programming Languages and Society, Ethics and Professionalism

Compared to CS 2013 which had a total of 24 CS core hours (tier-1 hours plus 80% of tier-2 hours), and 4 KA core hours (20% of tier-2 hours), the current recommendation has a total of 22 CS core hours and 20 KA core hours. Note that there is no requirement that each computer science graduate sees any KA core hours – they are a recommendation of content to consider if a program choses to offer greater emphasis on this knowledge area.

Note:

- Several topics within this knowledge area either build on, or overlap with, content covered in other knowledge areas such as 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.
- Different programming paradigms correspond to different problem domains. Most languages have evolved to integrate more than one programming paradigms such imperative with OOP, functional programming with OOP, logic programming with OOP, and event and reactive modeling with OOP. Hence, the emphasis is not on just one programming paradigm but a balance of all major programming paradigms.
- While the number of CS core and KA core hours is identified for each major programming paradigm (object-oriented, functional, logic), the distribution of hours across the paradigms may differ depending on the curriculum and

programming languages students have been exposed to leading up to coverage of this knowledge area. This document makes the assumption that students have exposure to a object-oriented programming language leading into this knowledge area.

- Imperative programming is not listed as a separate paradigm to be examined, instead it is treated as a subset of the object-oriented paradigm.
- With multicore computing, cloud computing, and computer networking becoming commonly available in the market, it has become imperative to understand the integration of “Distribution, concurrency, parallelism” along with other programming paradigms as a core area. This paradigm is integrated with almost all other major programming paradigms.
- With ubiquitous computing and real-time temporal computing becoming more in daily human life such as health, transportation, smart homes, it has become important to cover the software development aspects of event-driven and reactive programming, as well as parallel and distributed computing, under the programming languages knowledge area. Some of the topics covered will require, and interface with, concepts covered in knowledge areas such as Architecture and Organization, Operating Systems, and Systems Fundamentals.
- Some topics from the Parallel and Distributed Computing Knowledge Unit are likely to be integrated within the curriculum with topics from the Parallel and Distributed Programming Knowledge Area.
- There is an increasing interest in formal methods to prove program correctness and other properties. To support this, increased coverage of topics related to formal methods is included, but all of these topics are identified as non-core.
- When introducing these topics, it is also important that an instructor provides context for this material including why we have an interest in programming languages, and what they do for us in terms of providing a human readable version of instructions to a computer to execute for us.

Core Hours

Knowledge Units	**CS Core**	**KA Core**
Object-Oriented Programming	5	1
Functional Programming	4	3
Logic Programming		3
Compiled vs Interpreted Languages	1	
Scripting	2	
Event-Driven and Reactive Programming	2	2
Parallel and Distributed Computing	3	2
Type Systems	3	4
Systems Execution and Memory Model	3	
Language Translation and Execution		2
Program Abstraction and Representation		3

Syntax Analysis		
Compiler Semantic Analysis		
Program Analysis and Analyzers		
Code Generation		
Runtime Behavior and Systems		
Advanced Programming Constructs		
Language Pragmatics		
Formal Semantics		
Formal Development Methodologies		
Design Principles of Programming Languages		
Quantum Computing		
FPL and SEP		
Total	**23**	**20**

Knowledge Units

FPL-OOP: Object-Oriented Programming

CS Core:

1. Imperative programming as a sunset if object-oriented programming
2. Object-oriented design
    a. Decomposition into objects carrying state and having behavior
    b. Class-hierarchy design for modeling
3. Definition of classes: fields, methods, and constructors (See also: SDF-Fundamentals)
4. Subclasses, inheritance (including multiple inheritance), and method overriding
5. Dynamic dispatch: definition of method-call
6. Exception handling (See also: PDC-Coordination, SF-System-Reliability)
7. Object-oriented idioms for encapsulation
    a. Privacy, data hiding, and visibility of class members
    b. Interfaces revealing only method signatures
    c. Abstract base classes, traits and mixins
8. Dynamic vs static properties
9. Composition vs inheritance
10. Subtyping
    a. Subtype polymorphism; implicit upcasts in typed languages
    b. Notion of behavioral replacement: subtypes acting like supertypes
    c. Relationship between subtyping and inheritance

KA Core:

11. Collection classes, iterators, and other common library components

Illustrative Learning Outcomes:

CS Core:

1. Enumerate the difference between the imperative and object-oriented programming paradigms.
2. Compose a class through design, implementation, and testing to meet behavioral requirements.
3. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses.
4. Predict and validate control flow in a program using dynamic dispatch.
5. Compare and contrast:
    c. the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and
    d. the object-oriented approach-defining a class for each data variant with the class definition providing a method for each operation.
    e. Understand both as defining a matrix of operations and variants. (See also: FPL-Functional)
6. Compare and contrast the benefits and costs/impact of using inheritance (subclasses) and composition (in particular how to base composition on higher order functions).
7. 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).
8. Use object-oriented encapsulation mechanisms such as interfaces and private members.
9. 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. (See also: FPL-Functional)

KA Core:

10. Use collection classes and iterators effectively to solve a problem.

FPL-Functional: Functional Programming

CS Core:

1. Lambda expressions and evaluation (See also: AL-Models, FPL-Formalism)
    a. Variable binding and scope rules (See also: SDF-Fundamentals)
    b. Parameter passing (See also: SDF-Fundamentals)
    c. Nested lambda expressions and reduction order
2. Effect-free programming
    a. Function calls have no side effects, facilitating compositional reasoning
    b. Immutable variables and data copying vs. reduction
    c. Use of recursion vs. loops vs. pipelining (map/reduce)

3. Processing structured data (e.g., trees) via functions with cases for each data variant
    a. Functions defined over compound data in terms of functions applied to the constituent pieces
    b. Persistent data structures
4. Using higher-order functions (taking, returning, and storing functions)

KA Core:

5. Function closures (functions using variables in the enclosing lexical environment)
    a. Basic meaning and definition - creating closures at run-time by capturing the environment
    b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
    c. Using a closure to encapsulate data in its environment
    d. Lazy versus eager evaluation

Non-Core:

6. Graph reduction machine and call-by-need
7. Implementing lazy evaluation
8. Integration with logic programming paradigm using concepts such as equational logic, narrowing, residuation and semantic unification (See also: FPL-Logic)
9. Integration with other programming paradigms such as imperative and object-oriented

Illustrative learning outcomes:
CS Core:

1. Develop basic algorithms that avoid assigning to mutable state or considering reference equality.
2. Develop useful functions that take and return other functions.
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. (See also: FPL-OOP)

KA Core:

4. Explain a simple example of lambda expression being implemented using a virtual machine, such as a SECD machine, showing storage and reclaim of the environment.
5. Correctly interpret variables and lexical scope in a program using function closures.
6. Use functional encapsulation mechanisms such as closures and modular interfaces.
7. Compare and contrast stateful vs stateless execution.

8. 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. (See also: FPL-OOP)

Non-Core:

9. Illustrate graph reduction using a λ-expression using a shared subexpression
10. Illustrate the execution of a simple nested λ-expression using an abstract machine, such as an ABC machine.
11. Illustrate narrowing, residuation and semantic unification using simple illustrative examples.
12. Illustrate the concurrency constructs using simple programming examples of known concepts such as a buffer being read and written concurrently or sequentially. (See also: FPL-OOP)

FPL-Logic: Logic Programming

KA Core:

1. Universal vs. existential quantifiers (See also: AI-LRR, MSF-Discrete-mathematics)
2. First order predicate logic vs. higher order logic (See also: AI-LRR, MSF-Discrete-mathematics:8)
3. Expressing complex relations using logical connectives and simpler relations
4. Definitions of Horn clause, facts, goals and subgoals
5. Unification and unification algorithm; unification vs. assertion vs expression evaluation
6. Mixing relations with functions {see also: MSF-Discrete-mathematics:1)
7. Cuts, backtracking and non-determinism
8. Closed-world vs. open-world assumptions

Non-Core:

9. Memory overhead of variable copying in handling iterative programs
10. Programming constructs to store partial computation and pruning search trees
11. Mixing functional programming and logic programming using concepts such as equational logic, narrowing, residuation and semantic unification (See also: FPL-Functional)
12. Higher-order, constraint and inductive logic programming (See also: AI-LRR)
13. Integration with other programming paradigms such as object-oriented programming
14. Advance programming constructs such as difference-lists, creating user defined data structures, set of, etc.

Illustrative learning outcomes:
KA Core:

1. Use a logic language to implement a conventional algorithm.

2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts.
3. Use a simple illustrative example to show correspondence between First Order Predicate Logic (FOPL) and logic programs using Horn clauses.
4. Use examples to illustrate unification algorithm and its role of parameter passing in query reduction.
5. Use simple logic programs interleaving relations, functions, and recursive programming such as factorial and Fibonacci numbers and simple complex relationships between entities, and illustrate execution and parameter passing using unification and backtracking.

Non-Core:

6. Illustrate computation of simple programs such as Fibonacci and show overhead of recomputation, and then show how to improve execution overhead.

FPL-Compile-Interpret: Compiled vs Interpreted Languages Programming

CS Core:

1. Execution models for compiled languages and interpreted languages
2. Use of intermediate code, e.g., Bytecode
3. Limitations and benefits of compiled and interpreted languages

Illustrative Learning Outcomes:

CS Core:

1. Explain and understand the differences between compiled and interpreted languages, including the benefits and limitations of each.

FPL-Scripting: Scripting

CS Core:

1. Error/exception handling
2. Piping (See also: AR-Organization, SF-Overview:6, OS-Process:5)
3. System commands (See also: SF-A)
    a. Interface with operating systems (See also: SF-Overview, OS-Principles:3)
4. Environment variables (See also: SF-A)
5. File abstraction and operators (See also: SDF-Fundamentals, OS-Files, AR-IO, SF-Resource)
6. Data structures, such as arrays and lists (See also: AL-Fundamentals, SDF-Fundamentals, SDF-DataStructures)
7. Regular expressions (See also: AL-Models)
8. Programs and processes (See also: OS-Process)
9. Workflow

Illustrative learning outcomes:
CS Core:
1. Create and execute automated scripts to manage various system tasks.
2. Solve various text processing problems through scripting

FPL-Event-Driven: Event-Driven and Reactive Programming

CS Core:
1. Procedural programming vs. reactive programming: advantages of reactive programming in capturing events
2. Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers (See also: GIT-Interaction, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)
3. Stateless and state-transition models of event-based programming
4. Canonical uses such as GUIs, mobile devices, robots, servers (See also: GIT-Interaction, GIT-Image Processing, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)

KA Core:
5. Using a reactive framework
    a. Defining event handlers/listeners
    b. Parameterization of event senders and event arguments
    c. Externally-generated events and program-generated events
6. Separation of model, view, and controller

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

KA Core:
3. Define and use a reactive framework.
4. Describe an interactive system in terms of a model, a view, and a controller.

FPL-Parallel: Parallel and Distributed Programming

CS Core:
1. Safety and liveness (See also: PDC-Evaluation)
    a. Race conditions (See also: OS-Concurrency:2)
    b. Dependencies/preconditions

    c. Fault models (OS-Faults)
    d. Termination {see also: PDC-Coordination)
2. Programming models (See also: PDC-Programs).
        a. One or more of the following:
    a. Actor models
    b. Procedural and reactive models
    c. Synchronous/asynchronous programming models
    d. Data parallelism
3. Properties {see also: PDC-Programs, PDC-Coordination)
    a. Order-based properties:
        i. Commutativity
        ii. Independence
    b. Consistency-based properties:
        i. Atomicity
        ii. Consensus
4. Execution control (See also: PDC-Coordination, SF-B)
    a. Async await
    b. Promises
    c. Threads
5. Communication and coordination (See also: OS-Process:5, PDC-Communication, PDC-Coordination)
    a. Message-passing
    b. Shared memory
    c. cobegin-coend
    d. Monitors
    e. Channels
    f. Threads
    g. Guards

KA Core:

6. Futures
7. Language support for data parallelism such as forall, loop unrolling, map/reduce
8. Effect of memory-consistency models on language semantics and correct code generation
9. Representational State Transfer Application Programming Interfaces (REST APIs)
10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing
11. Overheads of message passing
12. Granularity of program for efficient exploitation of concurrency.
13. Concurrency and other programming paradigms (e.g., functional).

Illustrative learning outcomes:

CS Core:

1. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result.
2. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, synchronization constructs, and data-parallelism primitives.
3. \Use a message-passing model to analyze a communication protocol.
4. Use synchronization constructions such as monitor/synchronized methods in a simple program.
5. Modeling data dependency using simple programming constructs involving variables, read and write.
6. Modeling control dependency using simple constructs such as selection and iteration.

KA Core:

7. Explain how REST API's integrate applications and automate processes.
8. Explain benefits, constraints and challenges related to distributed and parallel computing.

FPL-Types: Type Systems

CS Core:

1. A type as a set of values together with a set of operations
   a. Primitive types (e.g., numbers, Booleans) (See also: SDF-Fundamentals)
   b. Compound types built from other types (e.g., records, unions, arrays, lists, functions, references using set operations) (See also: SDF-DataStructures)
2. Association of types to variables, arguments, results, and fields
3. Type safety as an aspect of program correctness (See also: FPL-Formalism)
4. Type safety and errors caused by using values inconsistently given their intended types
5. Goals and limitations of static and dynamic typing
   a. Detecting and eliminating errors as early as possible
6. Generic types (parametric polymorphism)
   a. Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion
   b. Comparison of monomorphic and polymorphic types
   c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
   d. Generic parameters and typing
   e. Use of generic libraries such as collections
   f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
   g. Prescriptive vs. descriptive polymorphism
   h. Implementation models of polymorphic types
   i. Subtyping

KA Core:

7. Type equivalence: structural vs name equivalence
8. Complementary benefits of static and dynamic typing
   a. Errors early vs. errors late/avoided
   b. Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections
   c. Typing rules
      i. Rules for function, product, and sum types
   d. Avoid misuse of code vs. allow more code reuse
   e. Detect incomplete programs vs. allow incomplete programs to run
   f. Relationship to static analysis
   g. Decidability

Non-Core:

9. Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
10. Type checking
11. Subtyping (See also: FPL-OOP)
    a. Subtype polymorphism; implicit upcasts in typed languages
    b. Notion of behavioral replacement: subtypes acting like supertypes
    c. Relationship between subtyping and inheritance
12. Type safety as preservation plus progress
13. Type inference
14. Static overloading
15. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) (See also: FPL-Formalism)
16. Dependent types (universal quantification as dependent function, existential quantification as dependent product) (See also: FPL-Formalism)

Illustrative learning outcomes:
CS Core:

1. Describe, for both a primitive and a compound type, the values that have that type.
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.
3. Describe examples of program errors detected by a type system.
4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically.
5. Describe an example program that does not type-check in a particular language and yet would have no error if run.
6. Use types and type-error messages to write and debug programs.

KA Core:

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

Non-Core:

13. Define a type system precisely and compositionally.
14. For various foundational type constructors, identify the values they describe and the invariants they enforce.
15. Precisely describe the invariants preserved by a sound type system.
16. Prove type safety for a simple language in terms of preservation and progress theorems.
17. Implement a unification-based type-inference algorithm for a simple language.
18. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs.

FPL-Systems: Systems Execution and Memory Model

CS Core:

1. Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string
2. Direct, indirect, and indexed access to memory location
3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects
4. Abstract low-level machine with simple instruction, stack and heap to explain translation and execution
5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap (See also: AR-Memory, OS-Memory)
    a. Translating selection and iterative constructs to control-flow diagrams
    b. Translating control-flow diagrams to low level abstract code
    c. Implementing loops, recursion, and tail calls
    d. Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine
6. Memory management (See also: AR-Memory, OS-Memory)
    a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects
    b. Return from procedure as automatic deallocation mechanism for local data elements in the stack

    c. Manual memory management: allocating, de-allocating, and reusing heap memory
    d. Automated memory management: garbage collection as an automated technique using the notion of reachability
7. Green computing (See also: SEP-Sustainability)

Illustrative learning outcomes:
CS Core:
1. Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions.
2. 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.
3. Investigate, identify, and fix memory leaks and dangling-pointer dereferences.

FPL-Translation: Language Translation and Execution

CS Core:
1. Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
    a. BNF and extended BNF representation of context-free grammar
    b. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement
    c. Execution as native code or within a virtual machine
2. Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution

KA Core:
3. Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
4. Secure compiler development (See also: SEC-Foundation, SEC-Defense)

Illustrative learning outcomes:
CS Core:
1. Differentiate a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.).
2. Differentiate syntax and parsing from semantics and evaluation.
3. Use BNF and extended BNF to specify the syntax of simple constructs such as if-then-else, type declaration and iterative constructs for known languages such as C++ or Python.
4. Illustrate parse tree using a simple sentence/arithmetic expression.

5. Illustrate translation of syntax diagrams to BNF/extended BNF for simple constructs such as if-then-else, type declaration, iterative constructs, etc.
6. Illustrate ambiguity in parsing using nested if-then-else/arithmetic expression and show resolution using precedence order

Non-Core:

7. Discuss the benefits and limitations of garbage collection, including the notion of reachability.

FPL-Abstraction: Program Abstraction and Representation

KA Core:

1. BNF and regular expressions
2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators
3. Components of a language
    a. Definitions of alphabets, delimiters, sentences, syntax and semantics
    b. Syntax vs. semantics
4. Program as a set of non-ambiguous meaningful sentences
5. Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling (See also: SDF-Fundamental)
6. Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation
7. Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables
8. Scope rules: static vs. dynamic; visibility of variables; side-effects
9. Side-effects induced by nonlocal variables, global variables and aliased variables

Non-Core:

10. L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values (See also: SDF-A)
11. Environment vs. store and their properties
12. Data and control abstraction
13. Mechanisms for information exchange between program units such as procedures, functions and modules: nonlocal variables, global variables, parameter passing, import-export between modules
14. Data structures to represent code for execution, translation, or transmission
15. Low level instruction representation such as virtual machine instructions, assembly language, and binary representation (See also: AR-B, AR-C)

16. Lambda calculus, variable binding, and variable renaming (See also: AL-Models, FPL-Formalism)
17. Types of semantics: operational, axiomatic, denotational, behavioral; define and use abstract syntax trees; contrast with concrete syntax

Illustrative learning outcomes:
KA Core:
1. Illustrate the scope of variables and visibility using simple programs
2. Illustrate different types of parameter passing using simple pseudo programming language
3. Explain side-effect using global and nonlocal variables and how to fix such programs
4. Explain how programs that process other programs treat the other programs as their input data.
5. Describe a grammar and an abstract syntax tree for a small language.
6. Describe the benefits of having program representations other than strings of source code.
7. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator.

FPL-Syntax: Syntax Analysis

Non-Core:
1. Regular grammars vs. context-free grammars (See also: AL-Models)
2. Scanning and parsing based on language specifications
3. Lexical analysis using regular expressions
4. Tokens and their use
5. Parsing strategies including top-down (e.g., recursive descent, or LL) and bottom-up (e.g., LR or GLR) techniques.
    a. Lookahead tables and their application to parsing
6. Language theory
    a. Chomsky hierarchy (See also: AL-Models)
    b. Left-most/right-most derivation and ambiguity
    c. Grammar transformation
7. Parser error recovery mechanisms
8. Generating scanners and parsers from declarative specifications

Illustrative learning outcomes:
Non-Core:
1. Use formal grammars to specify the syntax of languages.
2. Illustrate the role of lookahead tables in parsing.
3. Use declarative tools to generate parsers and scanners.
4. Recognize key issues in syntax definitions: ambiguity, associativity, precedence.

FPL-Semantics: Compiler Semantic Analysis

Non-Core:

1. Abstract syntax trees; contrast with concrete syntax
2. Defining, traversing and modifying high-level program representations
3. Scope and binding resolution
4. Static semantics
    a. Type checking
    b. Define before use
    c. Annotation and extended static checking frameworks
5. L-values/R-values (See also: SDF-Fundamentals)
6. Call semantics
7. Types of parameter-passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations
8. Declarative specifications such as attribute grammars and their applications in handling limited context-base grammar

Illustrative learning outcomes:
Non-Core:

1. Describe an abstract syntax tree for a small language
2. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences.
3. Describe semantic analyses using an attribute grammar.

FPL-Analysis: Program Analysis and Analyzers

Non-Core:

4. Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment.
5. Undecidability and consequences for program analysis
6. Flow-insensitive analysis, such as type-checking and scalable pointer and alias analysis
7. Flow-sensitive analysis, such as forward and backward dataflow analyses
8. Path-sensitive analysis, such as software model checking and software verification
9. Tools and frameworks for implementing analyzers
10. Role of static analysis in program optimization and data dependency analysis during exploitation of concurrency (See also: FPL-Code)
11. Role of program analysis in (partial) verification and bug-finding (See also: FPL-Code)
12. Parallelization
    a. Analysis for auto-parallelization

b. Analysis for detecting concurrency bugs

Illustrative learning outcomes:
Non-Core:
1. Define useful program analyses in terms of a conceptual framework such as dataflow analysis.
2. Explain the difference between dataflow graph and control flow graph.
3. Explain why non-trivial sound program analyses must be approximate.
4. Argue why an analysis is correct (sound and terminating).
5. Explain why potential aliasing limits sound program analysis and how alias analysis can help.
6. Use the results of a program analysis for program optimization and/or partial program correctness.

FPL-Code: Code Generation

Non-Core:
1. Instruction sets (See also: AR-C)
2. Control flow
3. Memory management (See also: AR-D, OS-F)
4. Procedure calls and method dispatching
5. Separate compilation; linking
6. Instruction selection
7. Instruction scheduling (e.g., pipelining)
8. Register allocation
9. Code optimization as a form of program analysis (See also: FPL-Analysis)
10. Program generation through generative AI,

Illustrative learning outcomes:
Non-Core:
1. Identify all essential steps for automatically converting source code into assembly or other low-level languages.
2. Generate the low-level code for calling functions/methods in modern languages.
3. Discuss why separate compilation requires uniform calling conventions.
4. Discuss why separate compilation limits optimization because of unknown effects of calls.
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.

FPL-Run-Time: Run-time Behavior and Systems

Non-Core:

1. Process models using stacks and heaps to allocate and deallocate activation records and recovering environment using frame pointers and return addresses during a procedure call including parameter passing examples.
2. Schematics of code lookup using hash tables for methods in implementations of object-oriented programs
3. Data layout for objects and activation records
4. Object allocation in heap
5. Implementing virtual entities and virtual methods; virtual method tables and their application
6. Run-time behavior of object-oriented programs
7. Compare and contrast allocation of memory during information exchange using parameter passing and non-local variables (using chain of static links)
8. Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
9. Just-in-time compilation and dynamic recompilation
10. Interface to operating system (e.g., for program initialization)
11. Interoperability between programming languages including parameter passing mechanisms and data representation (See also: AR-B)
    a. Big Endian, little endian
    b. Data layout of composite data types such as arrays
12. Other common features of virtual machines, such as class loading, threads, and security checking
13. Sandboxing

Illustrative learning outcomes:

Non-Core:

1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead.
2. Discuss benefits and limitations of automatic memory management.
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.
4. Compare and contrast static allocation vs. stack-based allocation vs. heap-based allocation of data elements.
5. Explain why some data elements cannot be automatically deallocated at the end of a procedure/method call (need for garbage collection).
6. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation.
7. Discuss use of sandboxing in mobile code
8. Identify the services provided by modern language run-time systems.

FPL-Constructs: Advanced Programming Constructs

Non-Core:

1. Encapsulation mechanisms
2. Lazy evaluation and infinite streams
3. Compare and contrast lazy evaluation vs. eager evaluation
4. Unification vs. assertion vs. expression evaluation
5. Control Abstractions: Exception Handling, Continuations, Monads
6. Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
7. Metaprogramming: Macros, Generative programming, Model-based development
8. String manipulation via pattern-matching (regular expressions)
9. Dynamic code evaluation ("eval")
10. Language support for checking assertions, invariants, and pre/post-conditions
11. Domain specific languages, such as database languages, data science languages, embedded computing languages, synchronous languages, hardware interface languages
12. Massive parallel high performance computing models and languages

Illustrative learning outcomes:
Non-Core:

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

FPL-Pragmatics: Language Pragmatics

Non-Core:

1. Effect of technology needs and software requirements on programming language development and evolution
2. Problems domains and programming paradigm
3. Criteria for good programming language design
    a. Principles of language design such as orthogonality
    b. Defining control and iteration constructs
    c. Modularization of large software
4. Evaluation order, precedence, and associativity
5. Eager vs. delayed evaluation
6. Defining control and iteration constructs
7. External calls and system libraries

Illustrative learning outcomes:
Non-Core:

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

FPL-Formalism: Formal Semantics

Non-Core:

1. Syntax vs. semantics
2. Approaches to semantics: Axiomatic, Operational, Denotational, Type-based.
3. Axiomatic semantics of abstract constructs such as assignment, selection, iteration using pre-condition, post-conditions and loop invariance
4. Operational semantics analysis of abstract constructs and sequence of such as assignment, expression evaluation, selection, iteration using environment and store
    a. Symbolic execution
    b. Constraint checkers
5. Denotational semantics
    a. Lambda Calculus (See also: AL-Models, FPL-Functional)
6. Proofs by induction over language semantics
7. Formal definitions and proofs for type systems (See also: FPL-Types)
    a. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum)
    b. Dependent types (universal quantification as dependent function, existential quantification as dependent product)
    c. Parametricity

Illustrative learning outcomes:
Non-Core:

1. Construct a formal semantics for a small language.
2. Write a lambda-calculus program and show its evaluation to a normal form.
3. Discuss the different approaches of operational, denotational, and axiomatic semantics.
4. Use induction to prove properties of all programs in a language.
5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system.
6. Use parametricity to establish the behavior of code given only its type.

FPL-Methodologies: Formal Development Methodologies

1. Formal specification languages and methodologies
2. Theorem provers, proof assistants, and logics
3. Constraint checkers See also: FPL-Formalism)
4. Dependent types (universal quantification as dependent function, existential quantification as dependent product) (See also: FPL-Types, FPL-Formalism)
5. Specification and proof discharge for fully verified software systems using pre/post conditions, refinement types, etc.
6. Formal modelling and manual refinement/implementation of software systems
7. Use of symbolic testing and fuzzing in software development
8. Model checking
9. Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

Illustrative learning outcomes:
Non-Core:
1. Use formal modeling techniques to develop and validate architectures.
2. Use proof assisted programming languages to develop fully specified and verified software artifacts.
3. Use verifier and specification support in programming languages to formally validate system properties.
4. Integrate symbolic validation tooling into a programming workflow.
5. Discuss when and how formal methods can be effectively used in the development process.

FPL-Design: Design Principles of Programming Languages

Non-core:
1. Language design principles
    a. simplicity
    b. security
    c. fast translation
    d. efficient object code
    e. orthogonality
    f. readability
    g. completeness
    h. implementation strategies
2. Designing a language to fit a specific domain or problem
3. Interoperability between programming languages,
4. Language portability
5. Formal description of s programming language
6. Green computing principles (See also: SEP-Sustainability)

Illustrative Learning Outcomes:
Non-core:
1. Understand what constitutes good language design and apply that knowledge to evaluate a real programming language.

FPL-Quantum: Quantum Computing

Non-core:
1. Advantages and disadvantages of quantum computing
2. Qubit and qubit state
3. superposition and interference
4. entanglement
5. Quantum algorithms (e.g., Shor's, Grover's algorithms)

Illustrative Learning Outcomes:
Non-core
1. An appreciation of quantum computing and its application to certain problems.
2. An appreciation of classical computing and its role as quantum computing emerges.

FPL-SEP: Society, Ethics and Professionalism

Non-Core:
1. Impact of English-centric programming languages
2. Enhancing accessibility and inclusivity for people with disabilities
   a. Supporting assistive technologies
3. Human factors related to programming languages and usability
   a. Impact of syntax on accessibility
   b. Supporting cultural differences (e.g., currency, decimals, dates)
   c. Neurodiversity
4. Etymology of terms such as "class", "master", "slave" in programming languages
5. Increasing accessibility by supporting multiple languages within applications (UTF)

Illustrative learning outcomes:
Non-Core:
1. Consciously design programming languages to be inclusive and non-offensive

Professional Dispositions

1. Meticulous: Students must demonstrate and apply the highest standards when using programming languages and formal methods to build safe systems that are fit for their purpose.

2. Meticulous: Attention to detail is essential when using programming languages and applying formal methods.
3. Inventive: Programming and approaches to formal proofs is inherently a creative process, students must demonstrate innovative approaches to problem solving. Students are accountable for their choices regarding the way a problem is solved.
4. Proactive: Programmers are responsible for anticipating all forms of user input and system behavior and to design solutions that address each one.
5. Persistent: Students must demonstrate perseverance since the correct approach is not always self-evident and a process of refinement may be necessary to reach the solution.

Math Requirements

Required:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).
- Mathematics – complex numbers, matrices, linear transformation, probability, statistics

Course Packaging Suggestions

Programming Language Concepts (Introduction) Course to include the following:

● FPL-OOP: Object-Oriented Programming	5 CS Core hours 1 KA Core hours
● FPL-Functional: Functional Programming	4 CS Core hours 3 KA Core hours
● FPL-Logic: Logic Programming	3 KA Core hours
● FPL-Compiled-Interpret: Compiled vs Interpreted Programming	1 CS Core hour
● FPL-Scripting: Scripting	2 CS Core hours
● FPL-Event-Driven: Event-Driven and Reactive Programming	2 CS Core hours 2 KA Core hours
● FPL-Parallel: Parallel and Distributed Computing	3 CS Core hours 2 KA Core hours
● FPL-Types: Type Systems	3 CS Core hours 4 KA Core hours

FPL-Systems: Systems Execution and Memory Model	3 CS Core hours
FPL-Translation: Language Translation and Execution	2 KA Core hours
FPL-Abstraction: Program Abstraction and Representation	3 KA Core hours
FPL-Quantum: Quantum Computing	2 CS Core hours
FPL-SEP: FPL and SEP	1 Non-Core hour

Pre-requisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar.

Programming Language Implementation (Advanced) Course to include the following:

FPL-Types: Type Systems	3 Non-core hours
FPL-Translation: Language Translation and Execution	3 Non-core hours
FPL-Syntax: Syntax Analysis	3 Non-core hours
FPL-Semantics: Compiler Semantic Analysis	5 Non-core hours
FPL-Analysis: Program Analysis and Analyzers	5 Non-core hours
FPL-Code: Code Generation	5 Non-core hours
FPL-Run-Time: Run-time Systems	4 Non-core hours
FPL-Constructs: Advanced Programming Constructs	4 Non-core hours
FPL-Pragmatics: Language Pragmatics	3 Non-core hours
FPL-Formalism: Formal Semantics	5 Non-core hours
FPL-Methodologies: Formal Development Methodologies	5 Non-core hours

Pre-requisites:

- Discrete mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar.
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).
- Introductory programming course.
- Programming proficiency in programming concepts such as:
  - type declarations such as basic data types, records, indexed data elements such as arrays and vectors, and class/subclass declarations, types of variables,
  - scope rules of variables,
  - selection and iteration concepts, function and procedure calls, methods, object creation
- Data structure concepts such as:

  - abstract data types, sequence and string, stack, queues, trees, dictionaries
  - pointer-based data structures such as linked lists, trees and shared memory locations
  - Hashing and hash tables
- System fundamentals and computer architecture concepts such as:
  - Digital circuits design, clocks, bus
  - registers, cache, RAM and secondary memory
  - CPU and GPU
- Basic knowledge of operating system concepts such as:
  - Interrupts, threads and interrupt-based/thread-based programming
  - Scheduling, including prioritization
  - Memory fragmentation
  - Latency

Competency Specifications

- **Task 1:** Select an appropriate programming paradigm based on the problem requirements.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Statement:** Apply knowledge of programming paradigms to select an appropriate paradigm(s) based on the problem and customer requirements.
- **Required knowledge areas and knowledge units:**
  - FPL-OOP: Object-Oriented Programming
  - FPL-Functional: Functional Programming
  - FPL-Logic: Logic Programming
  - FPL-Scripting: Scripting
  - FPL-Event-Driven: Event-Driven and Reactive Programming
  - FPL-Parallel: Parallel and Distributed Computing
- **Required skill level:** Evaluate
- **Core-level:** CS Core

- **Task 2:** Justify the choice of a programming paradigm to others (supervisor, peers, client).
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Statement:** Apply knowledge of programming paradigms to explain a choice of paradigm(s) based on the problem and customer requirements.
- **Required knowledge areas and knowledge units:**
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Scripting: Scripting
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - FPL-Parallel: Parallel and Distributed Computing
- **Required skill level:** Evaluate
- **Core-level:** CS Core

- **Task 3:** Select an appropriate programming language based on the programming paradigm chosen and the problem requirements.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Statement:** Apply knowledge of programming paradigms and programming languages to select an appropriate language(s) based on the chosen paradigm and problem and customer requirements.
- **Required knowledge areas and knowledge units:**
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Scripting: Scripting
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - FPL-Parallel: Parallel and Distributed Computing
- **Required skill level:** Evaluate
- **Core-level:** CS Core

- **Task 4:** Justify the choice of a programming language to others (supervisor, peers, client).
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Statement:** Apply knowledge of programming languages and paradigms to explain a choice of programming language(s) based on the selected paradigm and the problem and customer requirements.
- **Required knowledge areas and knowledge units:**
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Scripting: Scripting
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - FPL-Parallel: Parallel and Distributed Computing
- **Required skill level:** Evaluate
- **Core-level:** CS Core

- **Task 5**: Make an informed decision regarding which programming language/paradigm to select and use for a specific application.
- **Competency Statement**: Apply knowledge of multiple programming paradigms, including their strengths and weaknesses relative to the application to be developed, and select an appropriate paradigm and programming language.
- **Competency area**: Software/Application
- **Competency unit**: Evaluate/Develop
- **Required knowledge areas and knowledge units**:
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Event-Driven: Event-Driven and Reactive programming
    - FPL-Types: Type Systems
    - FPL-Translation: Language Translation and Execution
- **Required skill level: Explain/Evaluate**
- **Core-level: CS Core**

- **Task 6**: Use a scripting language to perform or automate a repetitive task.
- **Competency Statement:** Create and execute automated scripts to manage various system and development tasks.
- **Competency area:** Software/Systems/Application
- **Competency unit:** Development
- **Required knowledge areas and knowledge units:**
    - FPL-Scripting: Scripting
    - OS-Files: File Systems API and Implementation
- **Required skill level:** Develop
- **Core-level:** CS Core

- **Task 7:** Explain to a co-worker how a reactive or event-driven program works.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Statement:** Apply knowledge of event-driven and reactive programming to understand and explain the workings of a reactive or event-driven systems.
- **Required knowledge areas and knowledge units:**
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - SPD-Embedded: Embedded Platforms
- **Required skill level:** Evaluate
- · **Core-level:** CS Core

1. **Task 8:** Write a white paper which describes the benefits and challenges of a parallel or distributed program.
2. **Competency Statement:** Apply knowledge of parallel and distributed programming to determine enhancements/benefits over a sequential program.
3. **Competency area:** Software/Systems/Application/Theory
4. **Competency unit:** Design/Development
5. **Required knowledge areas and knowledge units:**
    - FPL-Parallel: Parallel and Distributed Programming
    - PDC-A: Programs and Execution
    - SF-Basics: Basic Concepts
6. **Required skill level:** Evaluate
7. **Core-level:** CS Core

- **Task 9:** Write a white paper to describe how a program is translated into machine code and executed.
- **Competency Statement:** Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions. 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. Be able to investigate, identify, and fix memory leaks and dangling-pointer dereferences.
- **Competency area:** Software/Systems
- **Competency unit:** Evaluation
- **Required knowledge areas and knowledge units:**
    - FPL-Translation: Language Translation and Execution
- **Required skill level:** Apply/Evaluate
- **Core-level:** CS Core

- **Task 10:** Effectively use a programming language's type system to develop safe and secure software.
- **Competency Statement:** Apply knowledge of static and dynamic type rules for a language to ensure an application is safe, secure, and correct.
- **Competency area:** Software/Application
- **Competency unit:** Development
- **Required knowledge areas and knowledge units:**
    - FPL-Types: Type Systems
- **Required skill level:** Develop
- **Core-level:** CS Core

- **Task 11:** Translate input from a high-level language into a lower-level form suitable for use by a computer (e.g., compiler; translate a natural language description in a game into instructions such as function calls).
- **Competency Statement:** Analyze input, determine its correctness and meaning, and translate into an alternative form appropriate for the application, possibly employing an intermediate form.
- **Competency area:** Software/Theory
- **Competency unit:** Design/Development
- **Required knowledge areas and knowledge units:**
    - FPL-H: Systems Programming
    - FPL-Translation: Language Translation and Execution
    - FPL-Abstraction: Program Abstraction and Representation
    - FPL-Syntax: Syntax Analysis
    - FPL-Semantics: Compiler Semantic Analysis
    - FPL-Analysis: Program Analysis and Analyzers
    - FPL-Code: Code Generation
    - FPL-Run-Time: Run-time Behavior and Systems
- **Required skill level:** Apply/Evaluate/Develop
- **Core-level:** KA Core

- **Task 12:** Discuss a program's correctness relative to a functional specification. Competency area: Software/Systems/Application/Theory
- **Competency Statement:** Utilize logic and formal methods to argue the correctness of a program.
- **Competency area:** Software/Systems/Application/Theory
- **Competency unit:** Development/Documentation/Acceptance
- **Required knowledge areas and knowledge units:**
    - FPL-Pragmatics: Language Pragmatics
    - FPL-Formalism: Formal Semantics
    - FPL-Methodologies: Formal Development Methodologies
- **Required skill level:** Explain
- **Core-level:** KA Core

- **Task 13:** Write a program using multiple languages and have the components interact effectively and efficiently with each other.
- **Competency Statement:** Apply knowledge of various programming paradigms and languages, and data and their implementation, as well as data representation, to develop a working multi-paradigm solution to a software problem.
- **Competency area:** Software/Application
- **Competency unit:** Design/Development
- **Required knowledge areas and knowledge units:**
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Scripting: Scripting
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - FPL-Parallel: Parallel and Distributed Computing
    - FPL-Types: Type Systems
    - FPL-Systems: Systems Programming
    - FPL-Translation: Language Translation and Execution
    - FPL-Abstraction: Program Abstraction and Representation
    - FPL-Constructs: Advanced Programming Constructs
    - FPL-Pragmatics: Language Pragmatics
- **Required skill level:** Explain/Apply/Evaluate/Develop
- **Core-level: KA Core**

- **Task 14:** Write a white paper explaining how a safe and secure program effectively utilized programming language features to make it safe and secure.
- **Competency Statement:** Apply knowledge of programming paradigms, type systems, static and dynamic semantics, and the compilation/interpretation process to explain how a program executes as it should and does so in a safe and efficient manner.
- **Competency area:** Software/Systems/Application
- **Competency unit:** Design/Development/Improvement
- **Required knowledge areas and knowledge units:**
  - FPL-OOP: Object-Oriented Programming
  - FPL-Functional: Functional Programming
  - FPL-Logic: Logic Programming
  - FPL-Scripting: Scripting
  - FPL-Event-Driven: Event-Driven and Reactive Programming
  - FPL-Parallel: Parallel and Distributed Computing
  - FPL-Types: Type Systems
  - FPL-Systems: Systems Programming
  - FPL-Translation: Language Translation and Execution
  - FPL-Abstraction: Program Abstraction and Representation
  - FPL-Constructs: Advanced Programming Constructs
  - FPL-Pragmatics: Language Pragmatics
- **Required skill level:** Apply/Evaluate
- **Core-level:** KA Core

- **Task 15:** Write a white paper explaining how a program executes in an efficient manner with respect to memory and CPU utilization.
- **Competency Statement:** Apply knowledge of programming paradigms, memory management, data representation and the compilation/interpretation process to explain how a program executes efficiently.
- **Competency area:** Software/Systems/Application
- **Competency unit:** Design/Development/Improvement
- **Required knowledge areas and knowledge units:**
    - FPL-OOP: Object-Oriented Programming
    - FPL-Functional: Functional Programming
    - FPL-Logic: Logic Programming
    - FPL-Scripting: Scripting
    - FPL-Event-Driven: Event-Driven and Reactive Programming
    - FPL-Parallel: Parallel and Distributed Computing
    - FPL-Systems: Systems Programming
    - FPL-Translation: Language Translation and Execution
    - FPL-Abstraction: Program Abstraction and Representation
    - FPL-Constructs: Advanced Programming Constructs
    - FPL-Pragmatics: Language Pragmatics
- **Required skill level:** Apply/Evaluate
- **Core-level:** KA Core

Committee

Chair: Michael Oudshoorn, High Point University, High Point, NC, USA
Members:

- Annette Bieniusa, TU Kaiserslautern, Kaiserslautern, Germany
- Brijesh Dongol, University of Surrey, Guildford, UK
- Michelle Kuttel, University of Cape Town, Cape Town, South Africa
- Doug Lea, State University of New York at Oswego, Oswego, NY, USA
- James Noble, Victoria University of Wellington, Wellington, New Zealand
- Mark Marron, Microsoft Research, Seattle, WA, USA and University of Kentucky, Lexington, KY, USA
- Peter-Michael Osera, Grinnell College, Grinnell, IA, USA
- Michelle Mills Strout, University of Arizona, Tucson, AZ, USA

Contributors:

- Alan Dearle, University of St. Andrews, St. Andrews, Scotland

Appendix: Core Topics and Skill Levels

Knowledge Unit	**Topic**	**Skill level**	**CS/KA Core**	**Hours**
Object-Oriented Programing	1. Imperative programming as a sunset if object-oriented programming 2. Object-oriented design a. Decomposition into objects carrying state and having behavior b. Class-hierarchy design for modeling 3. Definition of classes: fields, methods, and constructors 4. Subclasses, inheritance (including multiple inheritance), and method overriding 5. Dynamic dispatch: definition of method-call 6. Exception handling 7. Object-oriented idioms for encapsulation a. Privacy, data hiding, and visibility of class members b. Interfaces revealing only method signatures c. Abstract base classes, traits and mixins 8. Dynamic vs static properties 9. Composition vs inheritance 10. Subtyping a. Subtype polymorphism; implicit upcasts in typed languages b. Notion of behavioral replacement: subtypes acting like supertypes c. Relationship between subtyping and inheritance	Develop	CS	5
	11. Collection classes, iterators, and other common library components	Develop	KA	1

Functional Programming	1. Lambda expressions and evaluation a. Variable binding and scope rules b. Parameter passing c. Nested lambda expressions and reduction order 2. Effect-free programming a. Function calls have no side effects, facilitating compositional reasoning b. Immutable variables and data copying vs. reduction c. Use of recursion vs. loops vs. pipelining (map/reduce) 3. Processing structured data (e.g., trees) via functions with cases for each data variant a. Functions defined over compound data in terms of functions applied to the constituent pieces b. Persistent data structures 4. Using higher-order functions (taking, returning, and storing functions)	Develop	CS	4
	5. Function closures (functions using variables in the enclosing lexical environment) a. Basic meaning and definition - creating closures at run-time by capturing the environment b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments c. Using a closure to encapsulate data in its environment d. Lazy versus eager evaluation	Explain	KA	3

Logic Programming	1. Universal vs. existential quantifiers 2. First order predicate logic vs. higher order logic 3. Expressing complex relations using logical connectives and simpler relations 4. Definitions of Horn clause, facts, goals, and subgoals 5. Unification and unification algorithm; unification vs. assertion vs expression evaluation 6. Mixing relations with functions 7. Cuts, backtracking and non-determinism 8. Closed-world vs. open-world assumptions	Explain	KA	3
Scripting	1. Error/exception handling 2. Piping 3. System commands a. Interface with operating systems 4. Environment variables 5. File abstraction and operators 6. Data structures, such as arrays and lists 7. Regular expressions 8. Programs and processes 9. Workflow	Develop	CS	2
Event-driven and Reactive Programming	1. Procedural programming vs. reactive programming: advantages of reactive programming in capturing events 2. Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers 3. Behavior model of event-based programming 4. Canonical uses such as GUIs, mobile devices, robots, servers	Develop	CS	2

	5. Using a reactive framework a. Defining event handlers/listeners b. Parameterization of event senders and event arguments c. Externally-generated events and program-generated events 6. Separation of model, view, and controller	Develop	KA	2
Parallel and Distributed Computing	1. Safety and liveness a. Race conditions b. Dependencies/preconditions c. Fault models d. Termination 2. Programming models a. Actor models b. Procedural and reactive models c. Synchronous/asynchronous programming models d. Data parallelism 3. Semantics a. Commutativity b. Ordering c. Independence d. Consistency e. Atomicity f. Consensus 4. Execution control a. Async await b. Promises c. Threads 5. Communication and coordination a. Message-passing b. Shared memory c. cobegin-coend d. Monitors e. Channels f. Threads g. Guards	Develop	CS	3

	6. Futures 7. Language support for data parallelism such as forall, loop unrolling, map/reduce 8. Effect of memory-consistency models on language semantics and correct code generation 9. Representational State Transfer Application Programming Interfaces (REST APIs) 10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing 11. Overheads of message passing 12. Granularity of program for efficient exploitation of concurrency. 13. Concurrency and other programming paradigms (e.g., functional).	Explain	KA	2

Type Systems	1. A type as a set of values together with a set of operations a. Primitive types (e.g., numbers, Booleans) b. Compound types built from other types (e.g., records, unions, arrays, lists, functions, references) using set operations 2. Association of types to variables, arguments, results, and fields 3. Type safety as an aspect of program correctness 4. Type safety and errors caused by using values inconsistently given their intended types 5. Statically-typed vs dynamically-typed programming languages 6. Goals and limitations of static and dynamic typing a. Detecting and eliminating errors as early as possible 7. Generic types (parametric polymorphism) a. Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion b. Comparison of monomorphic and polymorphic types c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism d. Generic parameters and typing e. Use of generic libraries such as collections f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism g. Prescriptive vs. descriptive polymorphism h. Implementation models of polymorphic types	Develop	CS	3

	i. Subtyping			

	8. Type equivalence: structural vs name equivalence 9. Complementary benefits of static and dynamic typing a. Errors early vs. errors late/avoided b. Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections c. Typing rules i. Rules for function, product, and sum types d. Avoid misuse of code vs. allow more code reuse e. Detect incomplete programs vs. allow incomplete programs to run f. Relationship to static analysis g. Decidability	Develop	KA	4

Systems Programming	1. Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string 2. Direct, indirect, and indexed access to memory location 3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects 4. Abstract low-level machine with simple instruction, stack and heap to explain translation and execution 5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap a. Translating selection and iterative constructs to control-flow diagrams b. Translating control-flow diagrams to low level abstract code c. Implementing loops, recursion, and tail calls a. Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine 6. Memory management a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects b. Return from procedure as automatic deallocation mechanism for local data elements in the stack c. Manual memory management: allocating, de-allocating, and reusing heap memory d. Automated memory management: garbage collection as an	Develop	CS	3

	automated technique using the notion of reachability			

Language Translation and Execution	1. Interpretation vs. compilation to native code vs. compilation to portable intermediate representation a. BNF and extended BNF representation of context-free grammar b. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement c. Execution as native code or within a virtual machine 2. Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution	Explain	CS	4
	3. Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures) 4. Secure compiler development	Explain	KA	1

Program Abstraction and Representation	1. BNF and regular expressions 2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators 3. Components of a language a. Definitions of alphabets, delimiters, sentences, syntax and semantics b. Syntax vs. semantics 4. Program as a set of non-ambiguous meaningful sentences 5. Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling 6. Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation 7. Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables 8. Scope rules: static vs. dynamic; visibility of variables; side-effects 9. Side-effects induced by nonlocal variables, global variables and aliased variables	Explain	KA	3

Total CS Core hours: 23
Total KA Core hours: 20